

# CS 241: Systems Programming

## Lecture 23. Advanced Git

Fall 2023

Prof. Stephen Checkoway

# Branches

Visualize a project's development as initially a *linked list* of commits

When a development track splits, a new branch is created

- This gives us a *tree* of commits

In Git, a branch is actually just a pointer to a leaf in the tree of development

Two or more branches can be merged together

- This gives a *graph* of commits

# Using branches

Development and release versions

Trying out new features

Focusing on fixing a bug

Simpler to do in Git than other VCS, consider using more frequently

# Git branching

List all branches in the project

- `git branch`

Create a new branch

- `git branch <branchname>`

Switch to a branch

- `git checkout <branchname>`

Create and immediately switch

- `git checkout -b <branchname>`

Delete a branch

- `git branch -d <branchname>`

# Using branches

Create and switch to a branch

```
$ git branch working  
  
$ git checkout working  
M README  
Switched to branch 'working'  
  
$ git branch  
main  
* working
```

# Stashing

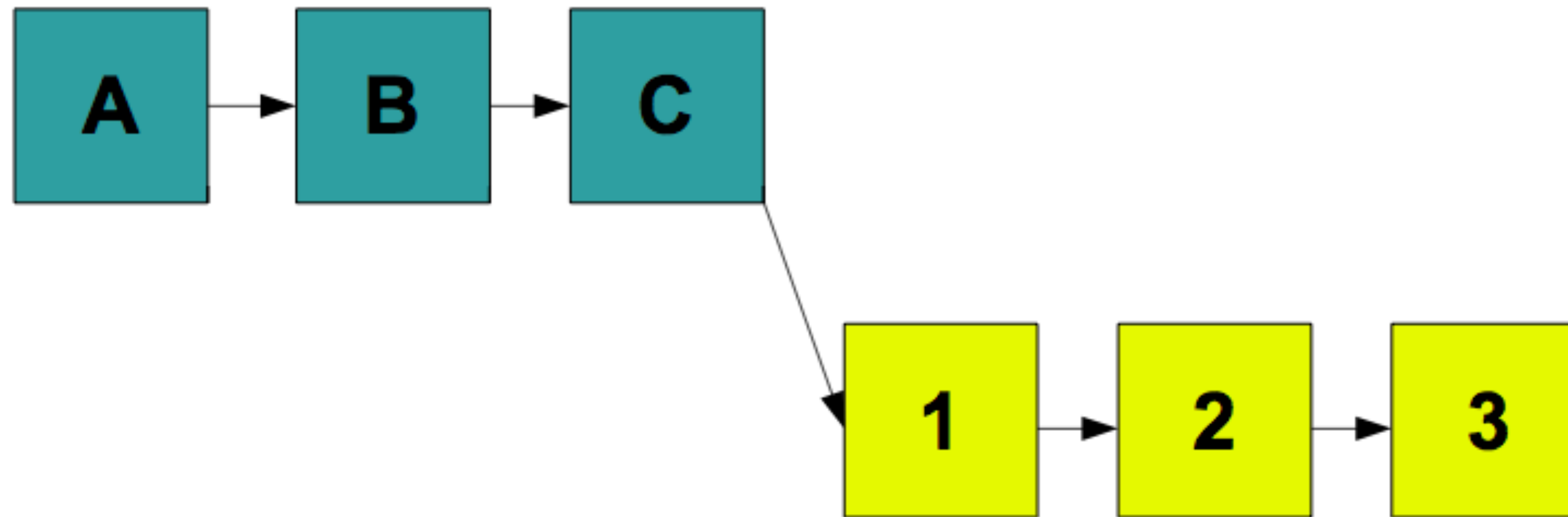
Working tree should be clean when switching branches

Save/hide changes you're not ready to commit with `git stash`

- ▶ Pushes changes onto a stash stack

Recover changes later with `git stash pop`

# Using branches



# Using branches

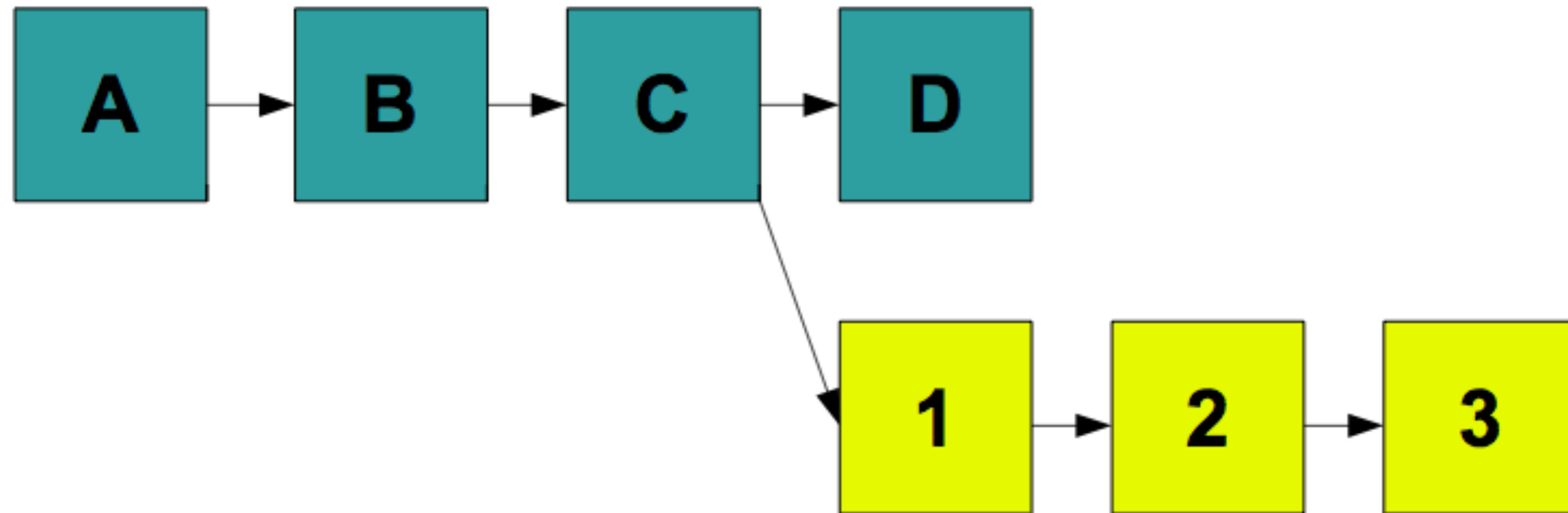
Integrate changes back into **main**

```
$ git checkout main
Switched to branch 'main'

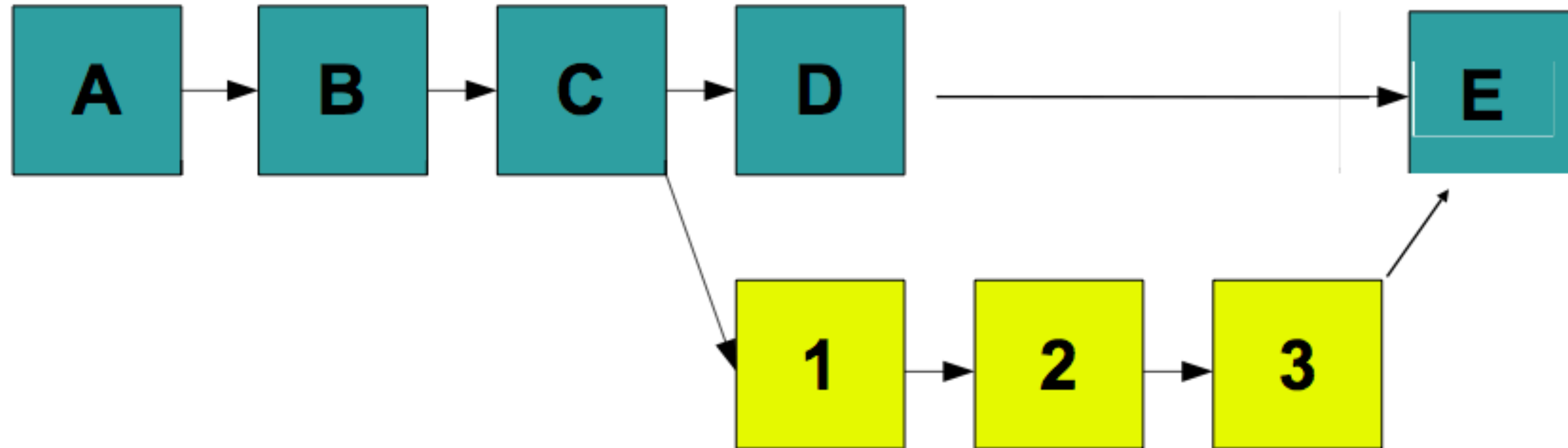
$ git merge working
Merge made by the 'recursive' strategy.
 newfile.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 newfile.txt
```



# Before git merge



# After git merge



# Merged history

```
*   cdd07b2 - (HEAD, main) Merge branch
'working'
| \
| * 1ccf9e7 - (working) Added a new file
* | 3637a76 - Second change
* | cf98d00 - First change
| /
* cf31a23 - Updated README to 2.0
* 2a8fc15 - Initial commit
```

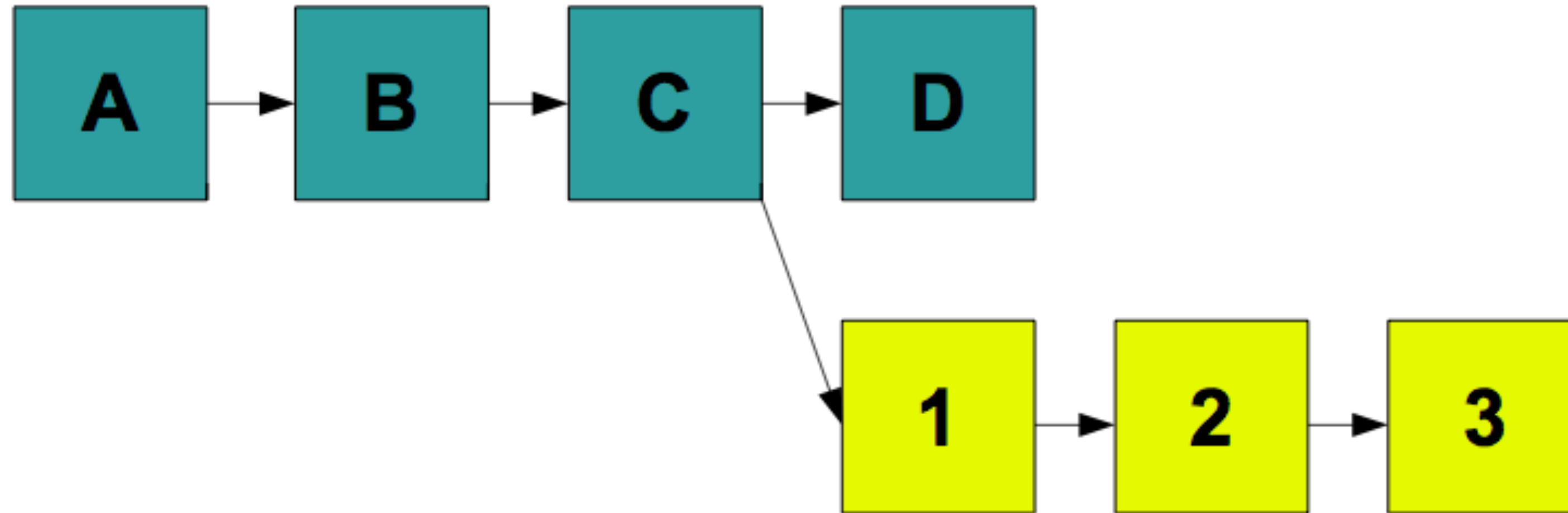
# Rebasing

Like merging, rebasing transfers changes from one branch to another

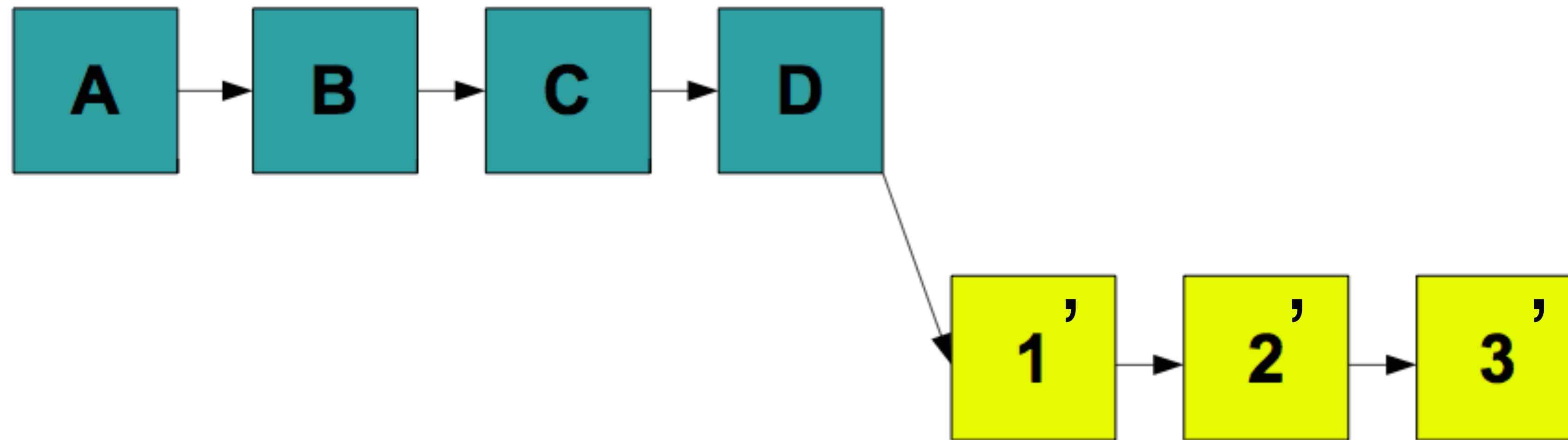
Does not create a new commit

Replays changes from current branch onto head of other branch

# Before git rebase



# After git rebase



# git rebase

Powerful tool

Can change the commit order

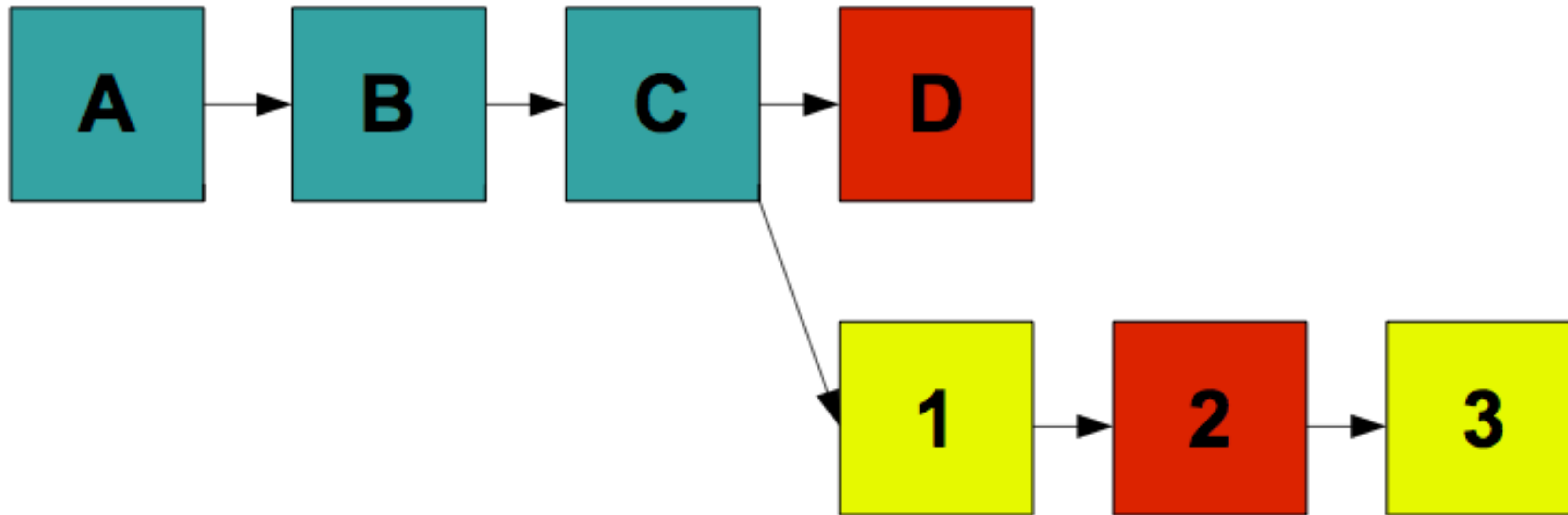
Merge/split commits

Make fixes in earlier commits

- DO NOT DO ON PUSHED CHANGES OR PUBLIC BRANCH

```
$ git rebase -i main
```

# Conflicts





# Git conflict markers


```
$ cat foo.c
<<<<<<< HEAD
current content
=====
branch content
>>>>>> newbranch
$ vim foo.c
$ git add foo.c
$ git rebase --continue
```

# Pull requests with Github

Contributing changes to repositories on Github

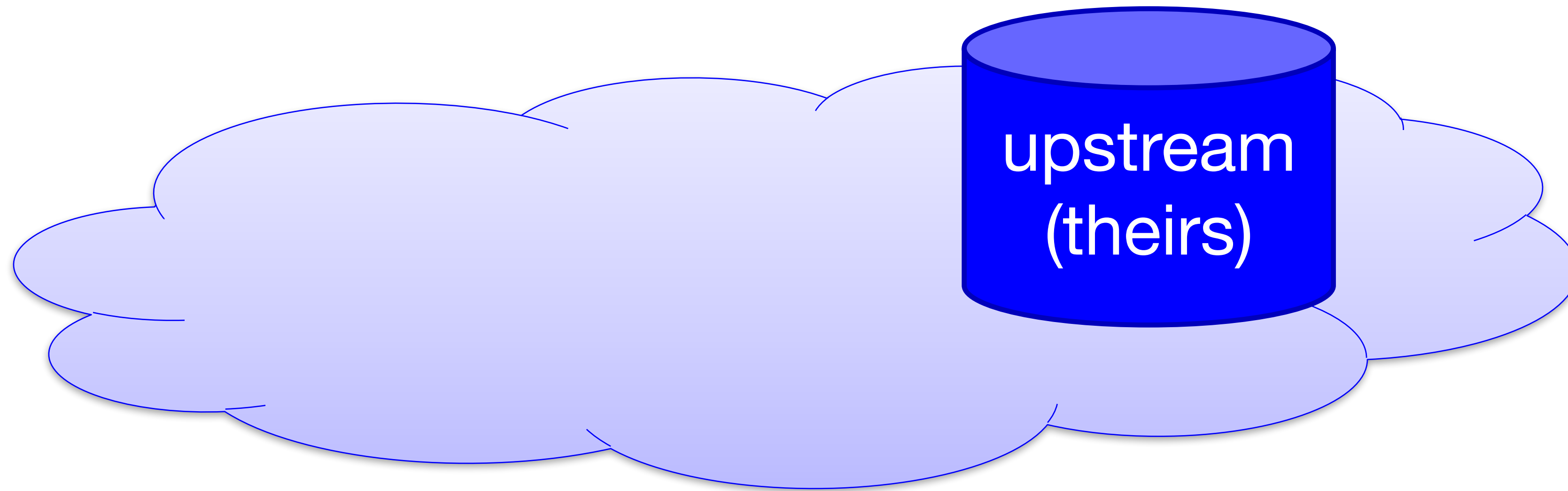
Requests the owner of the code integrate your changes

# Setup

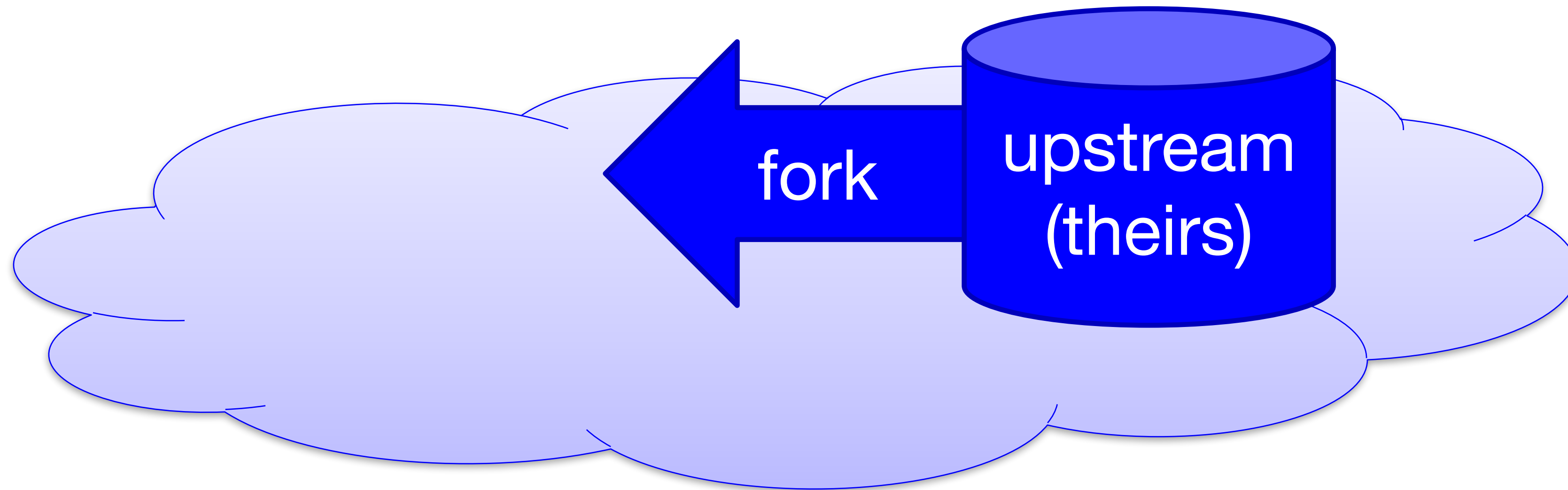


**GitHub**

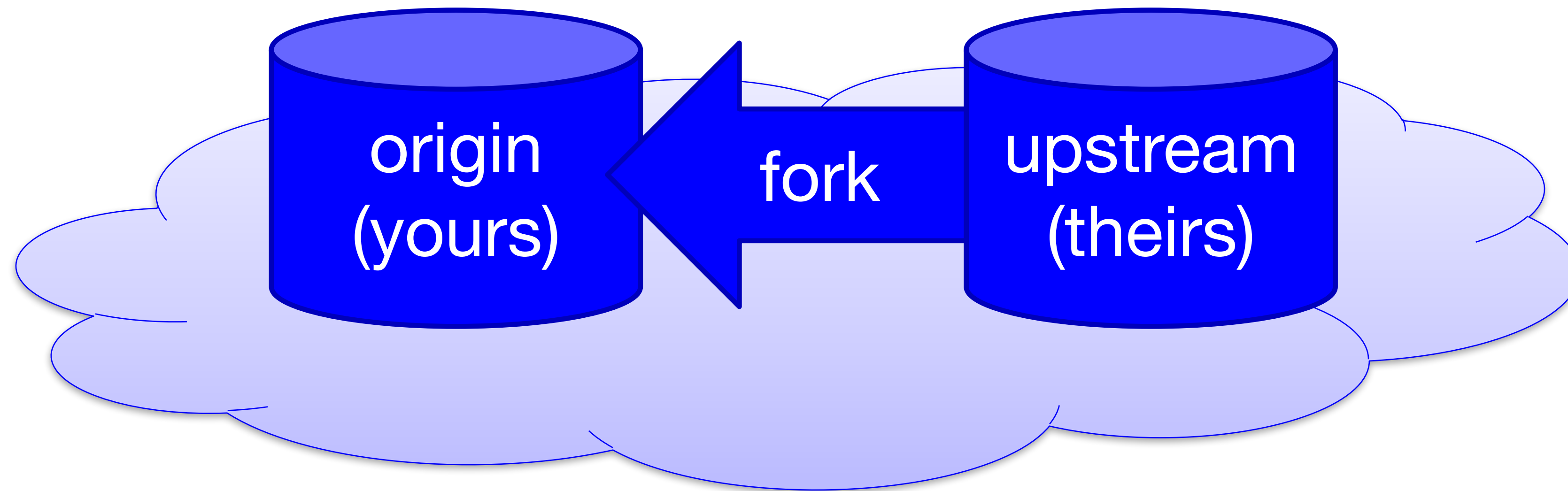
# Setup



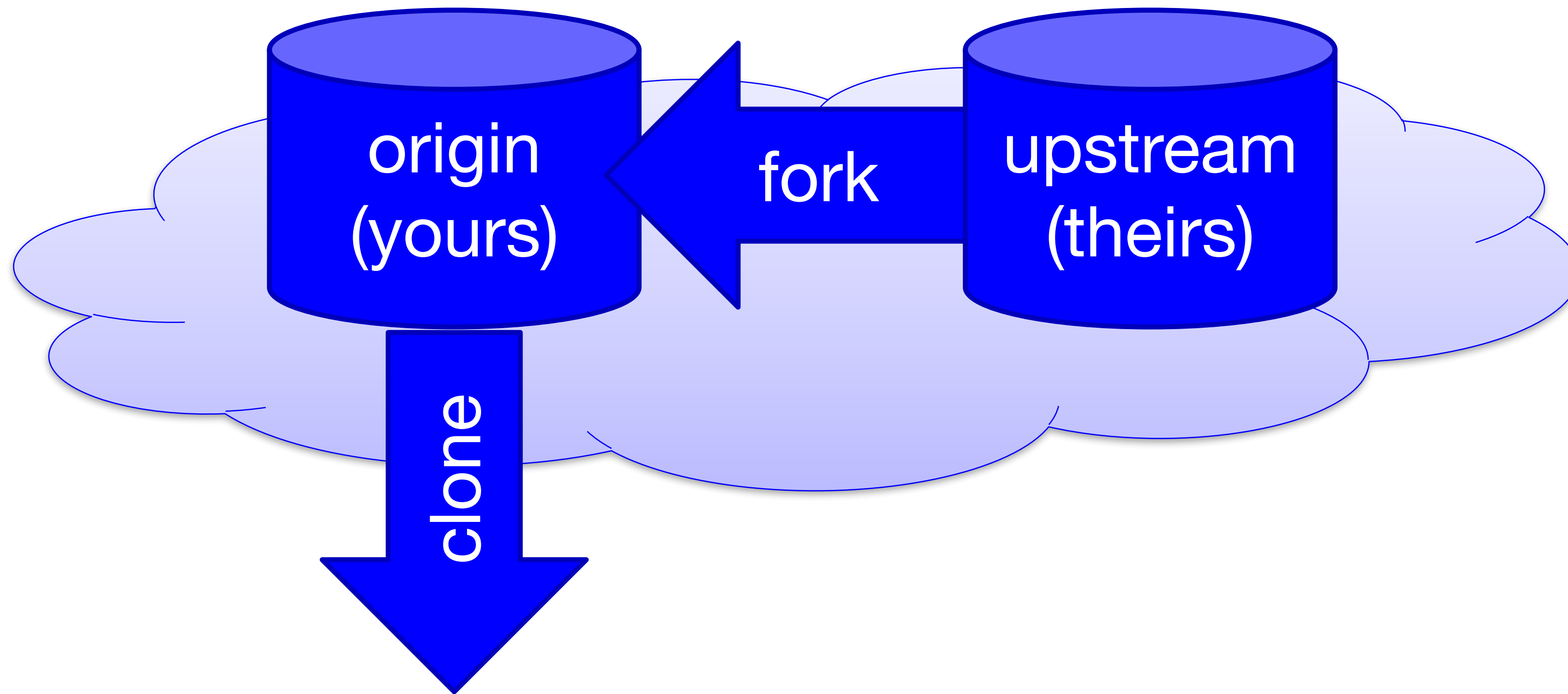
# Setup



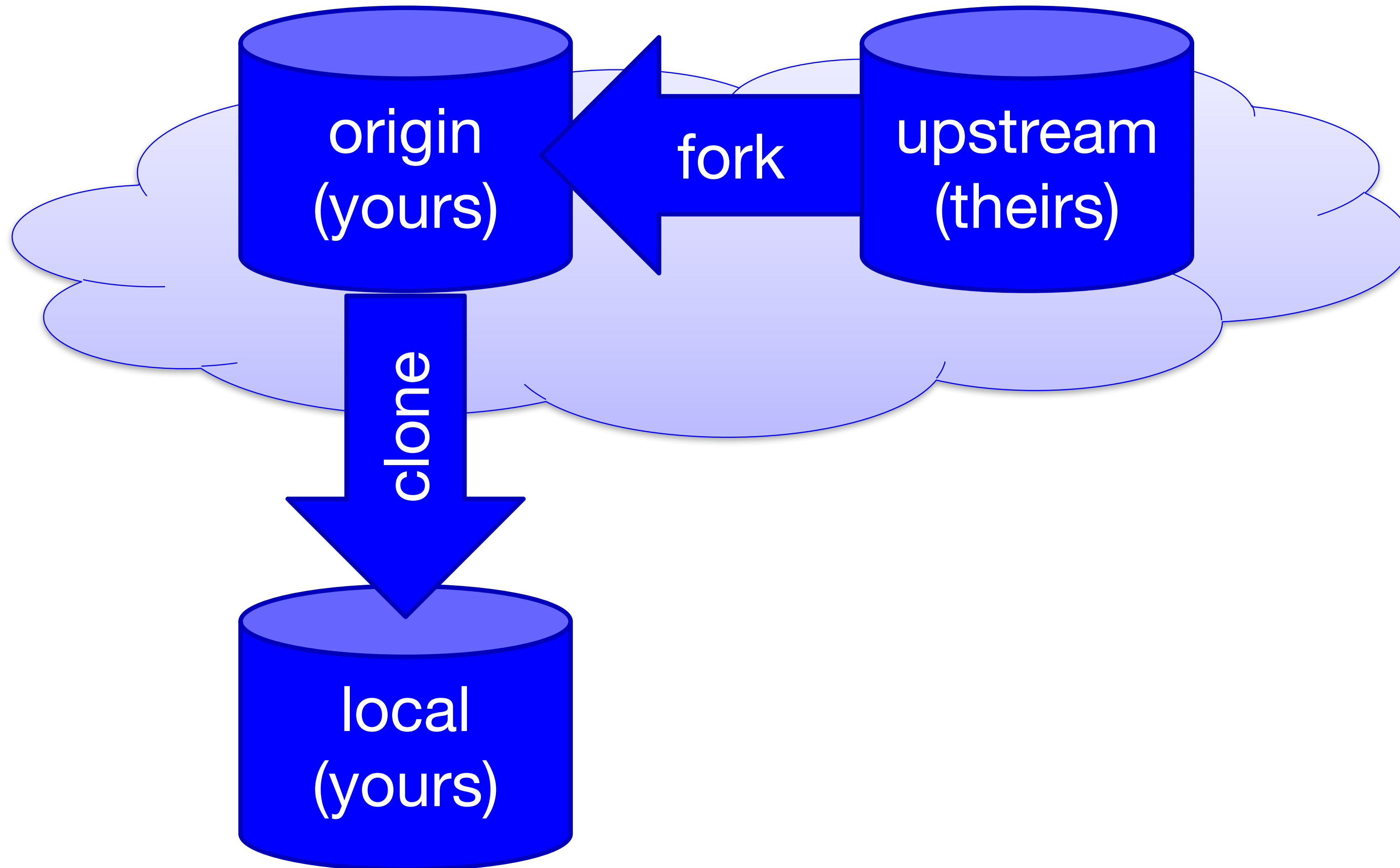
# Setup



# Setup

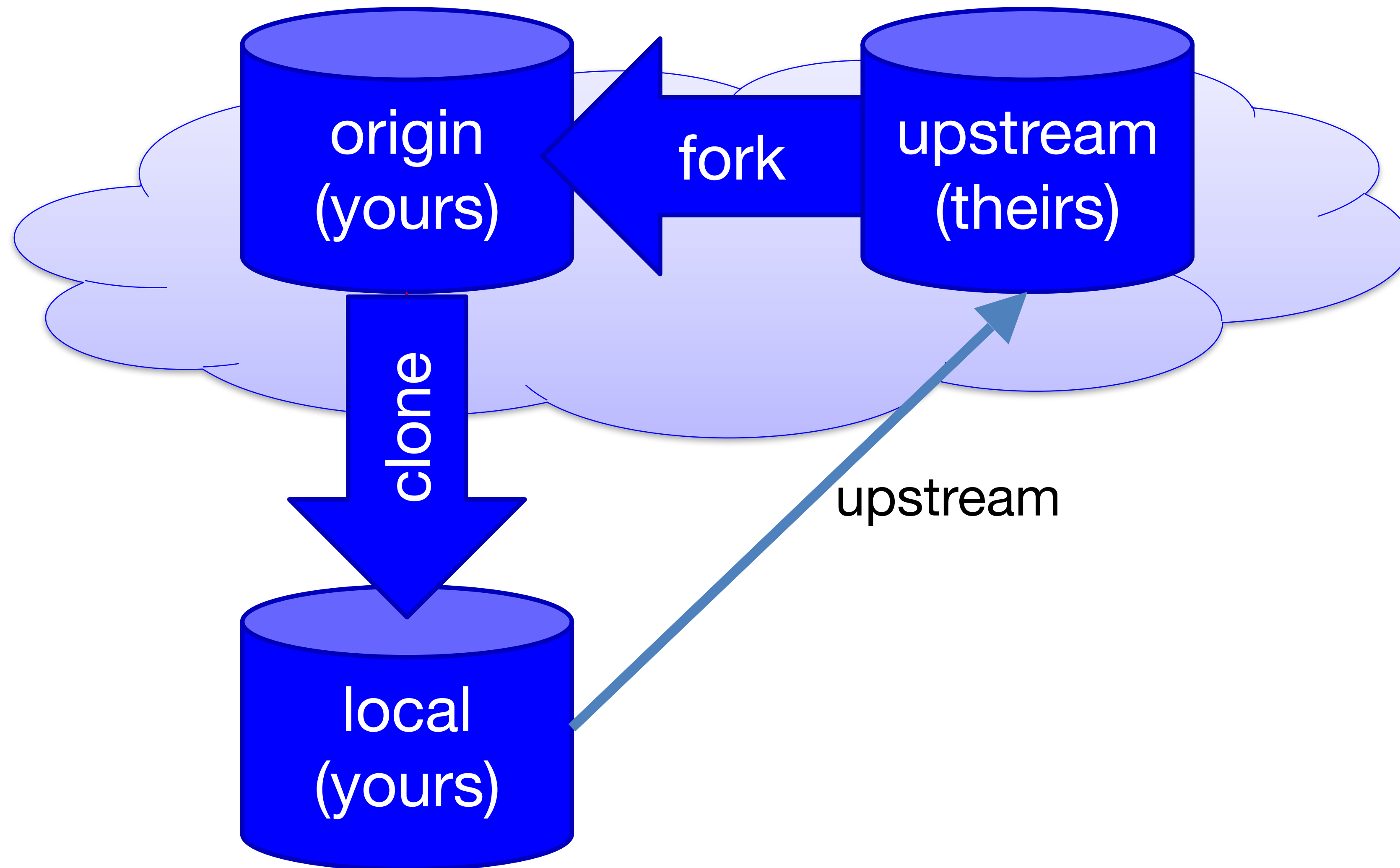


# Setup

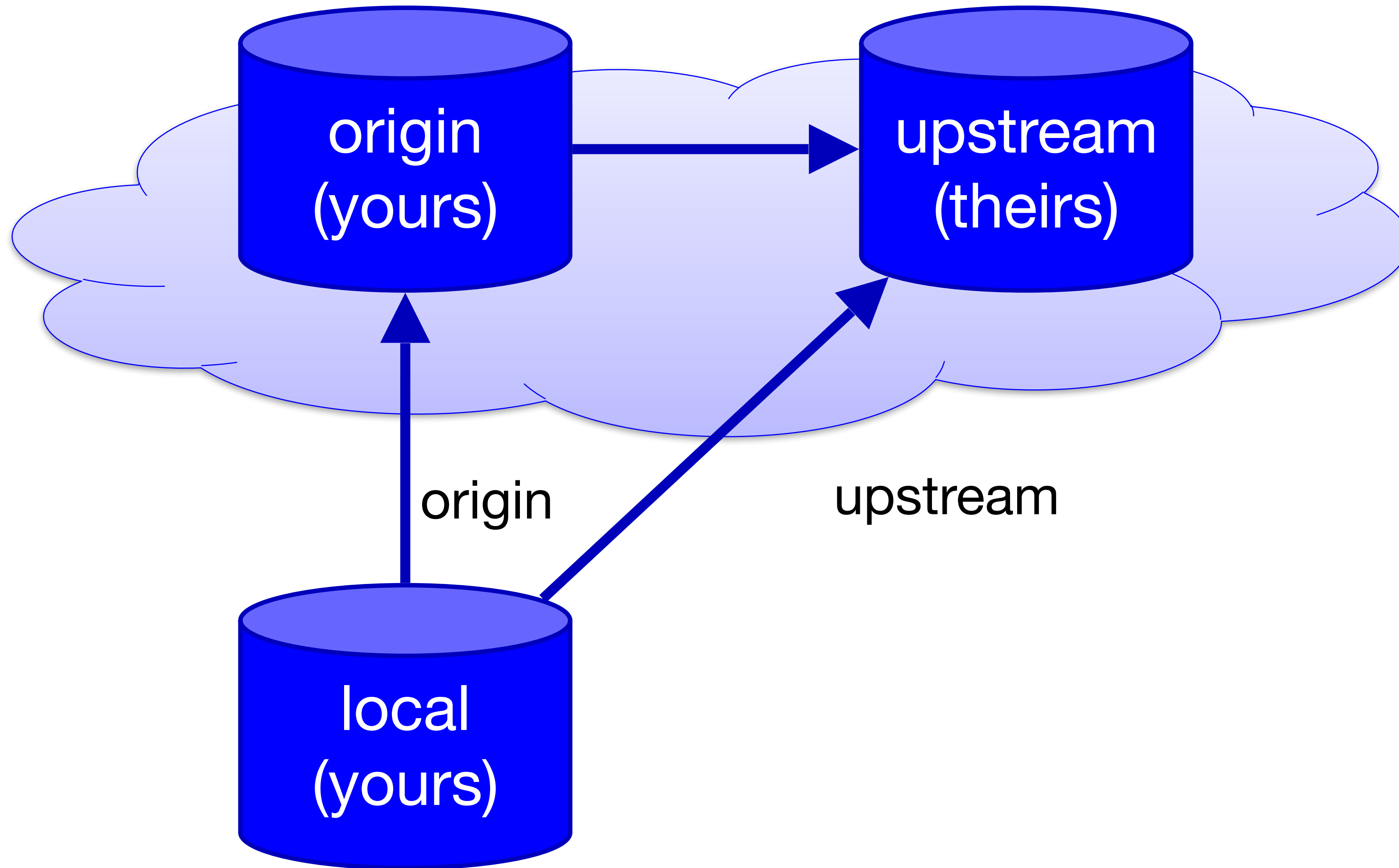




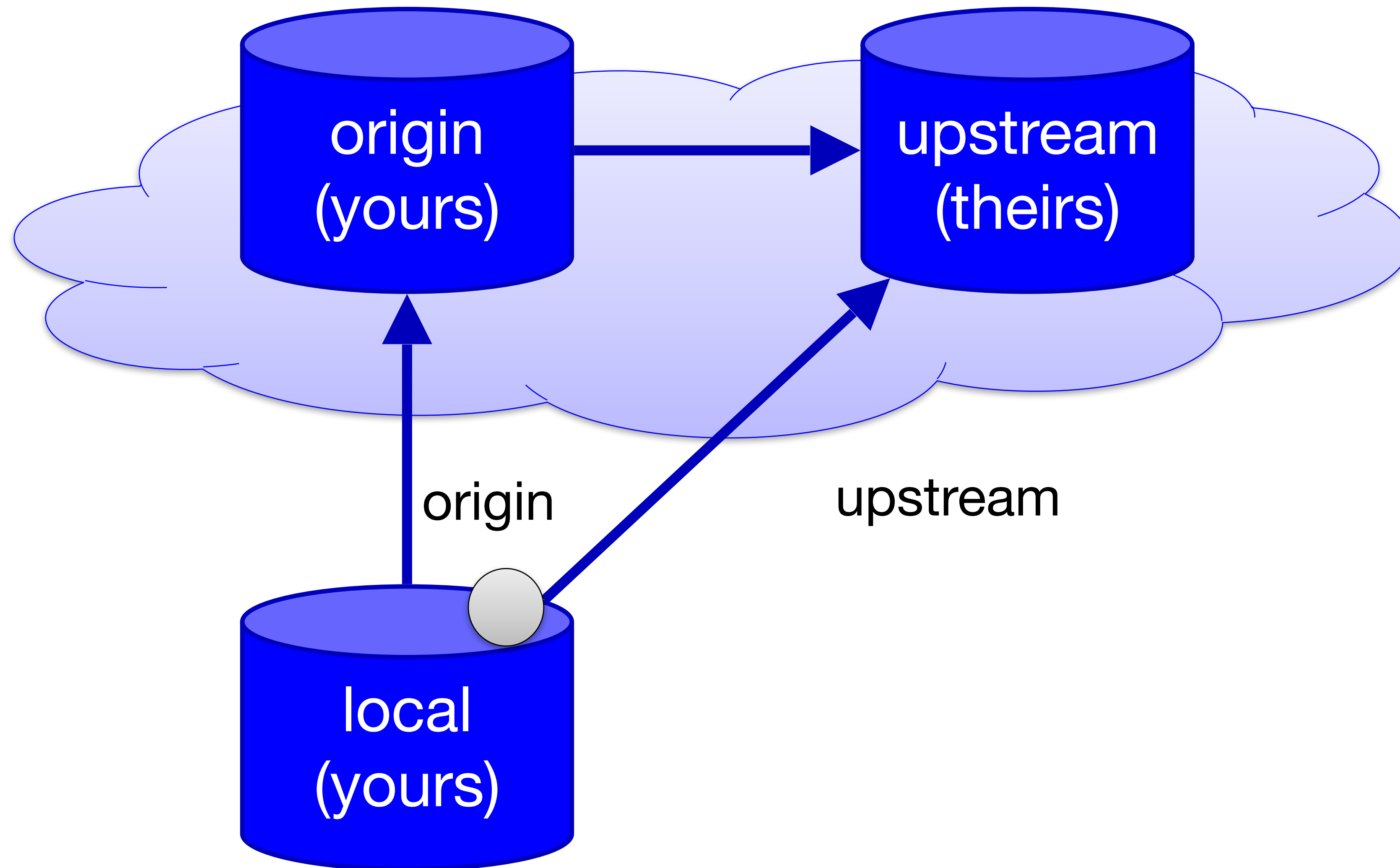
# Setup



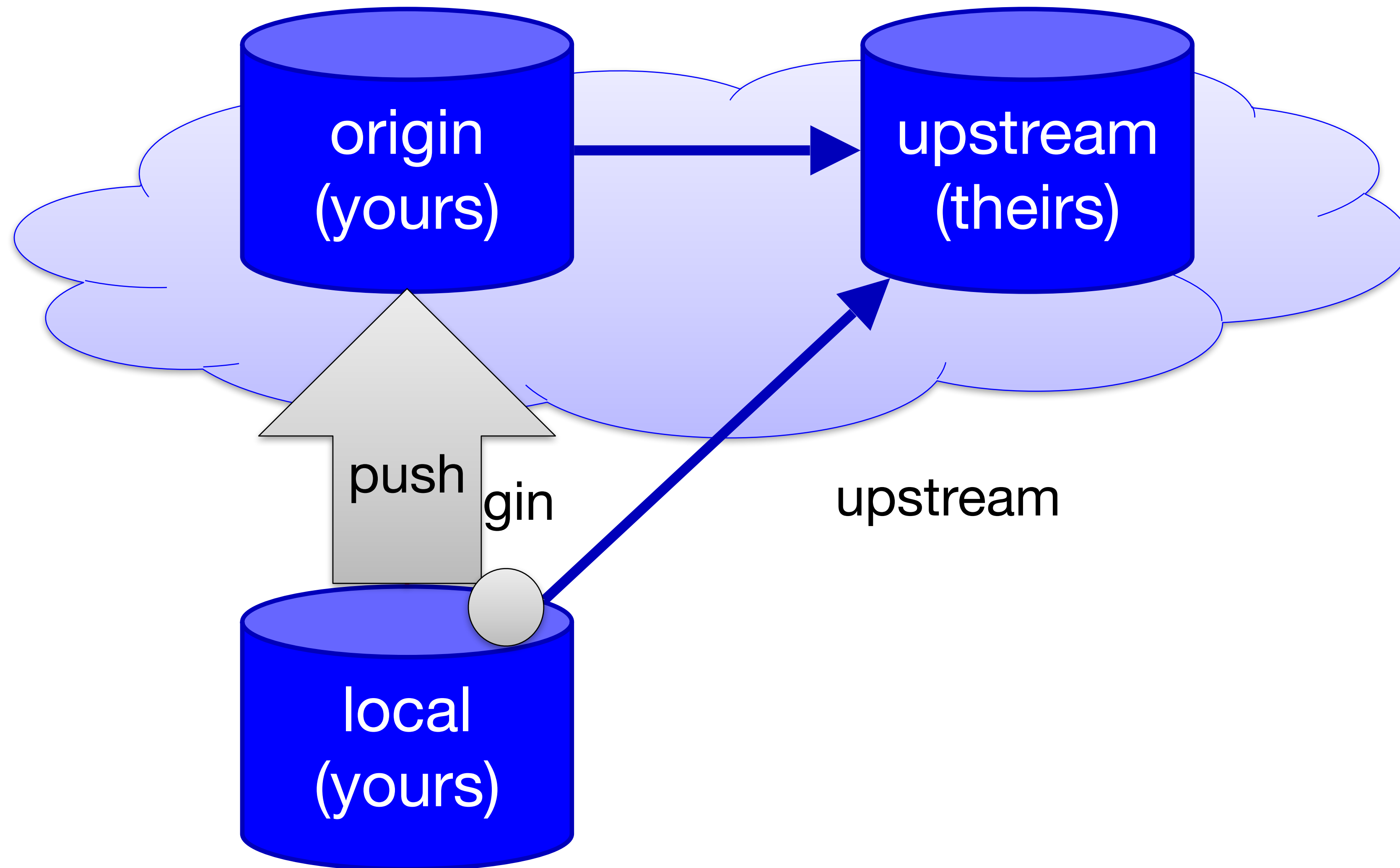
# Setup



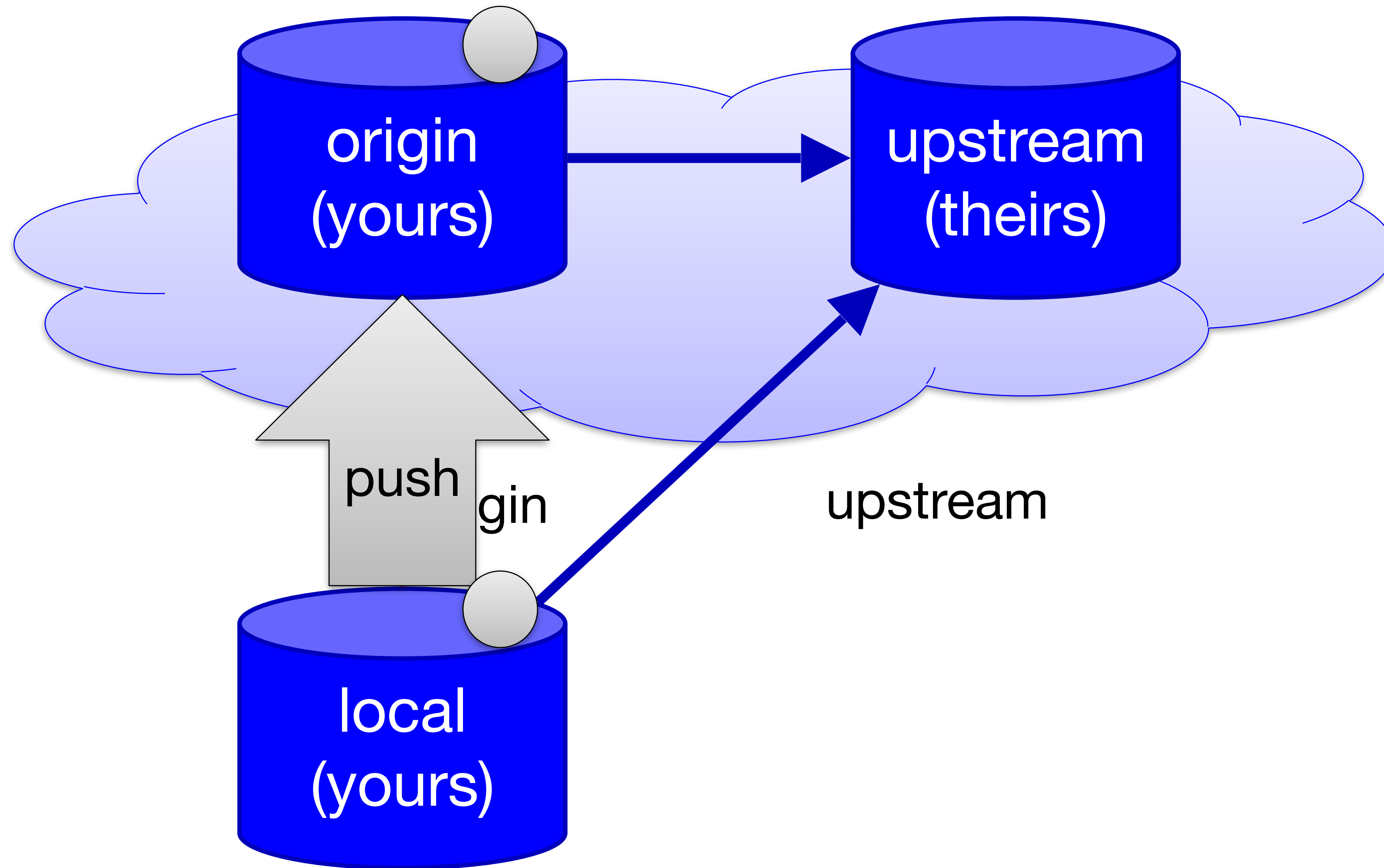
# Contribute Changes



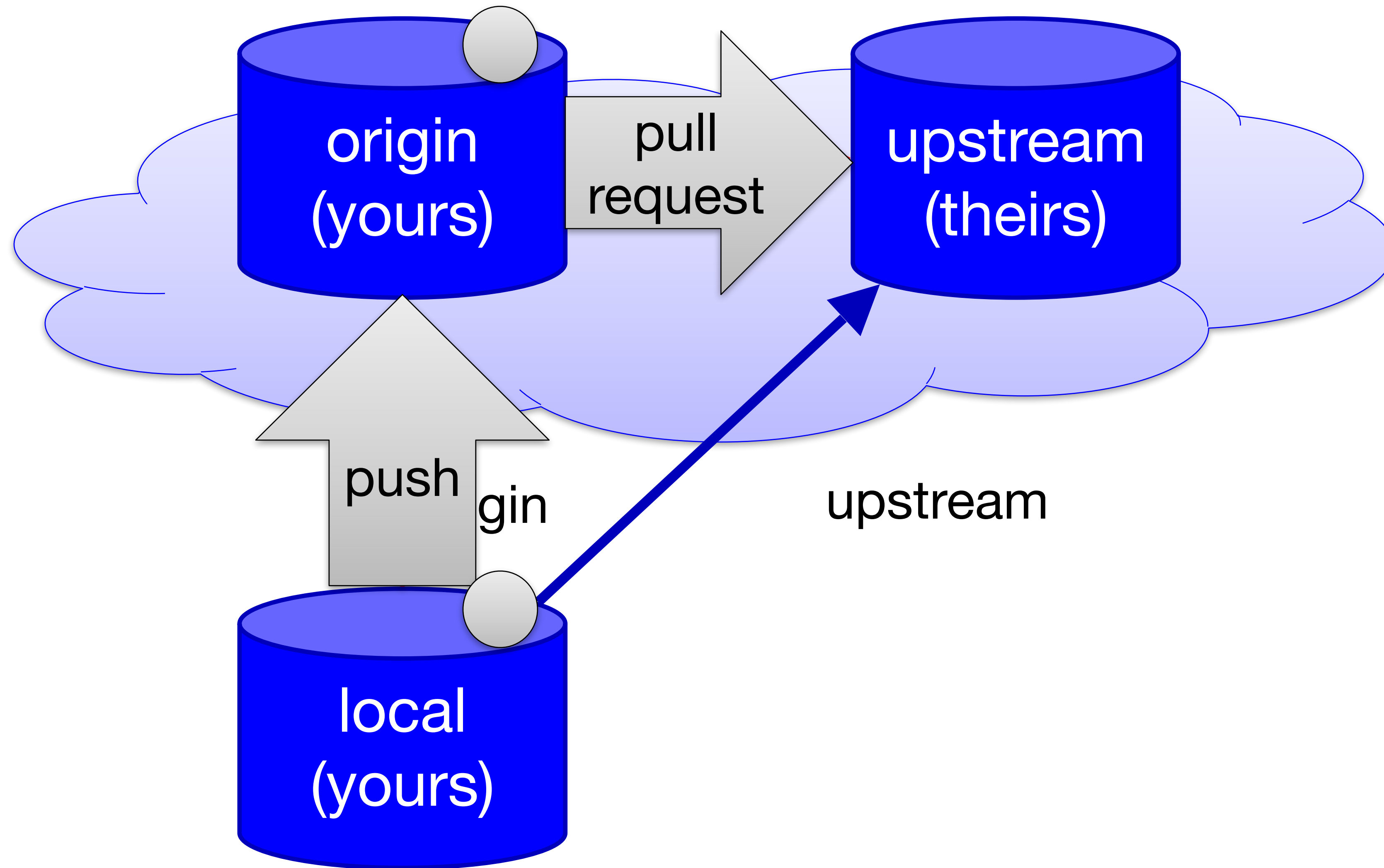
# Contribute Changes



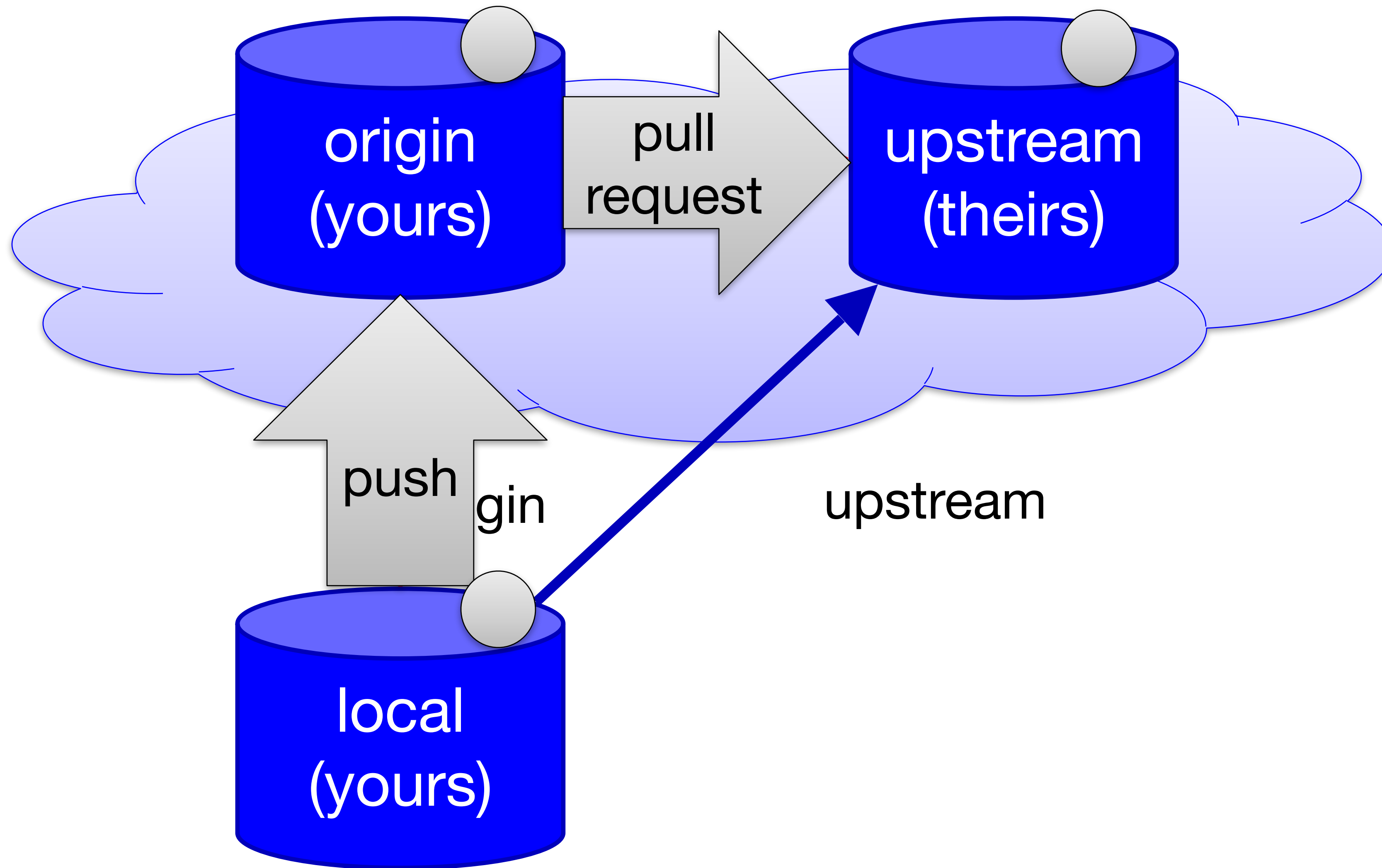
# Contribute Changes



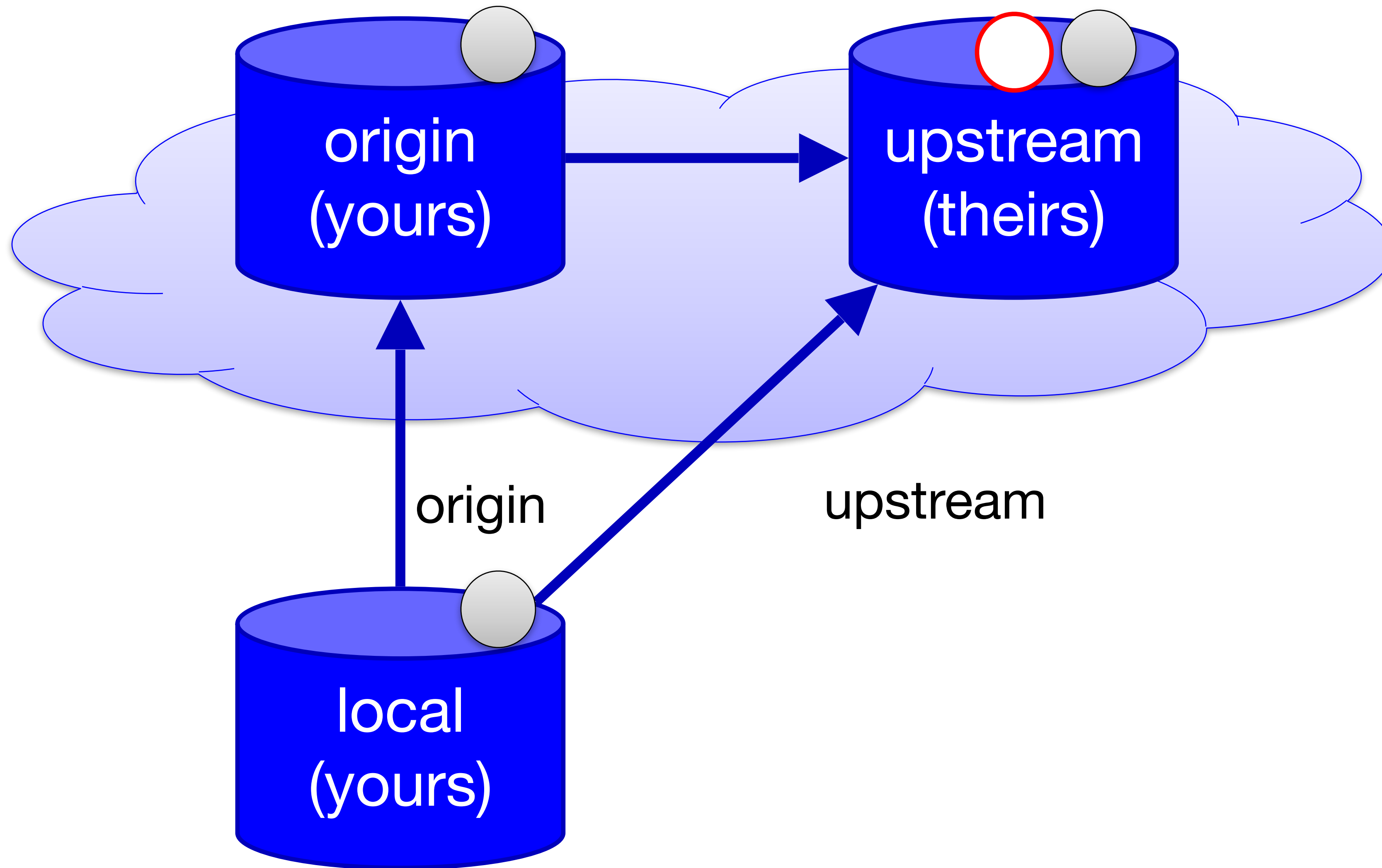
# Contribute Changes



# Contribute Changes

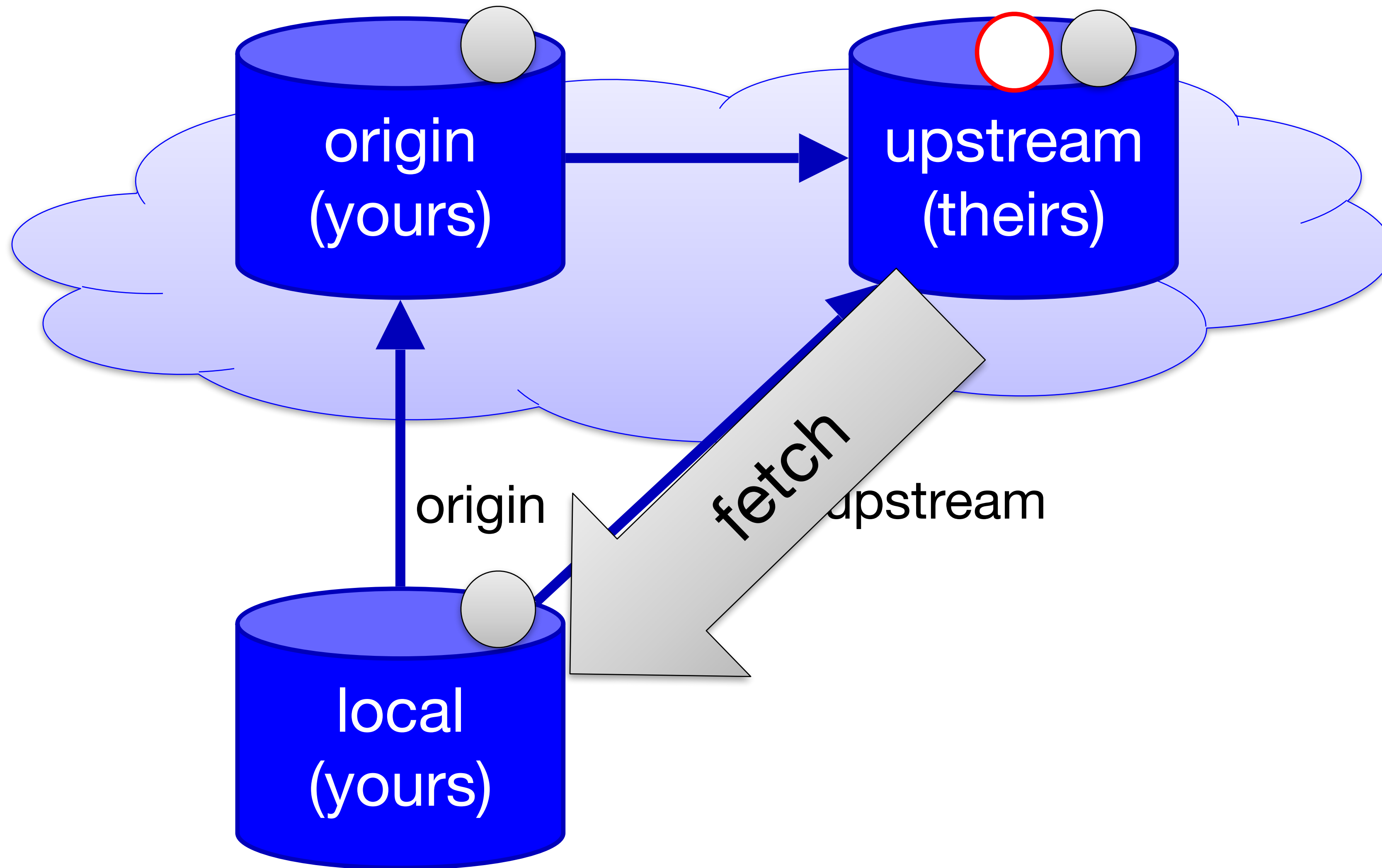


# Integrate Changes

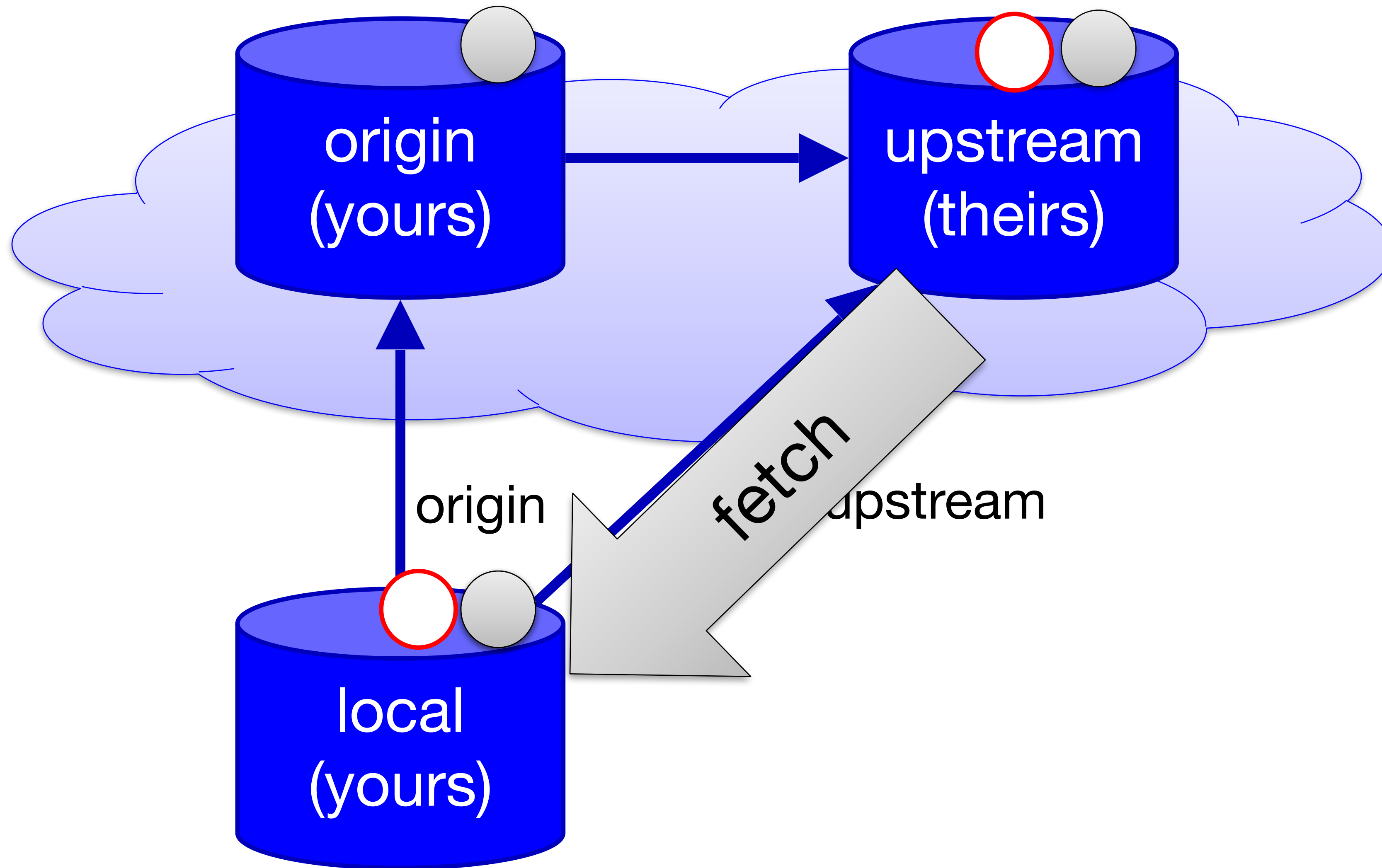




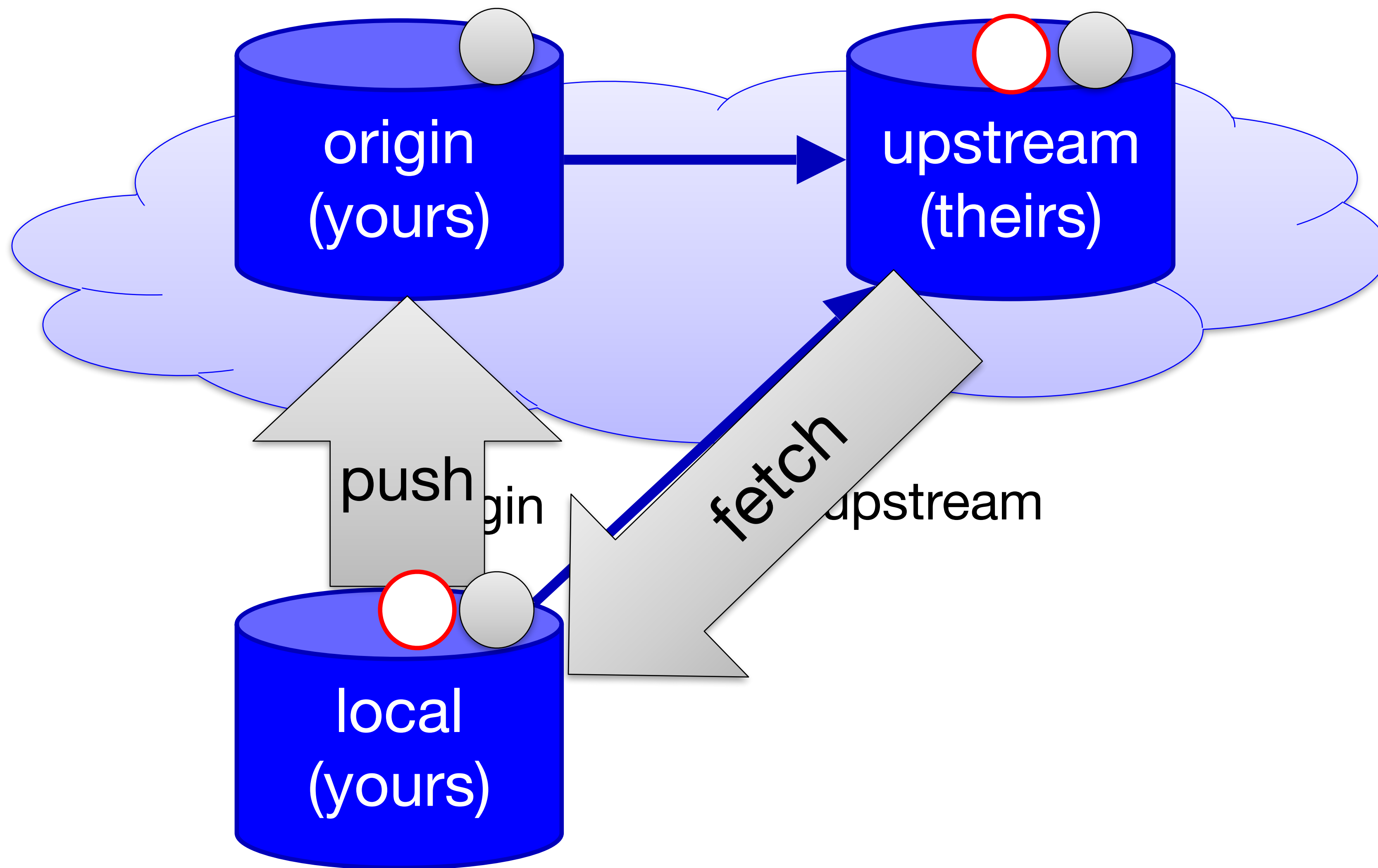
# Integrate Changes



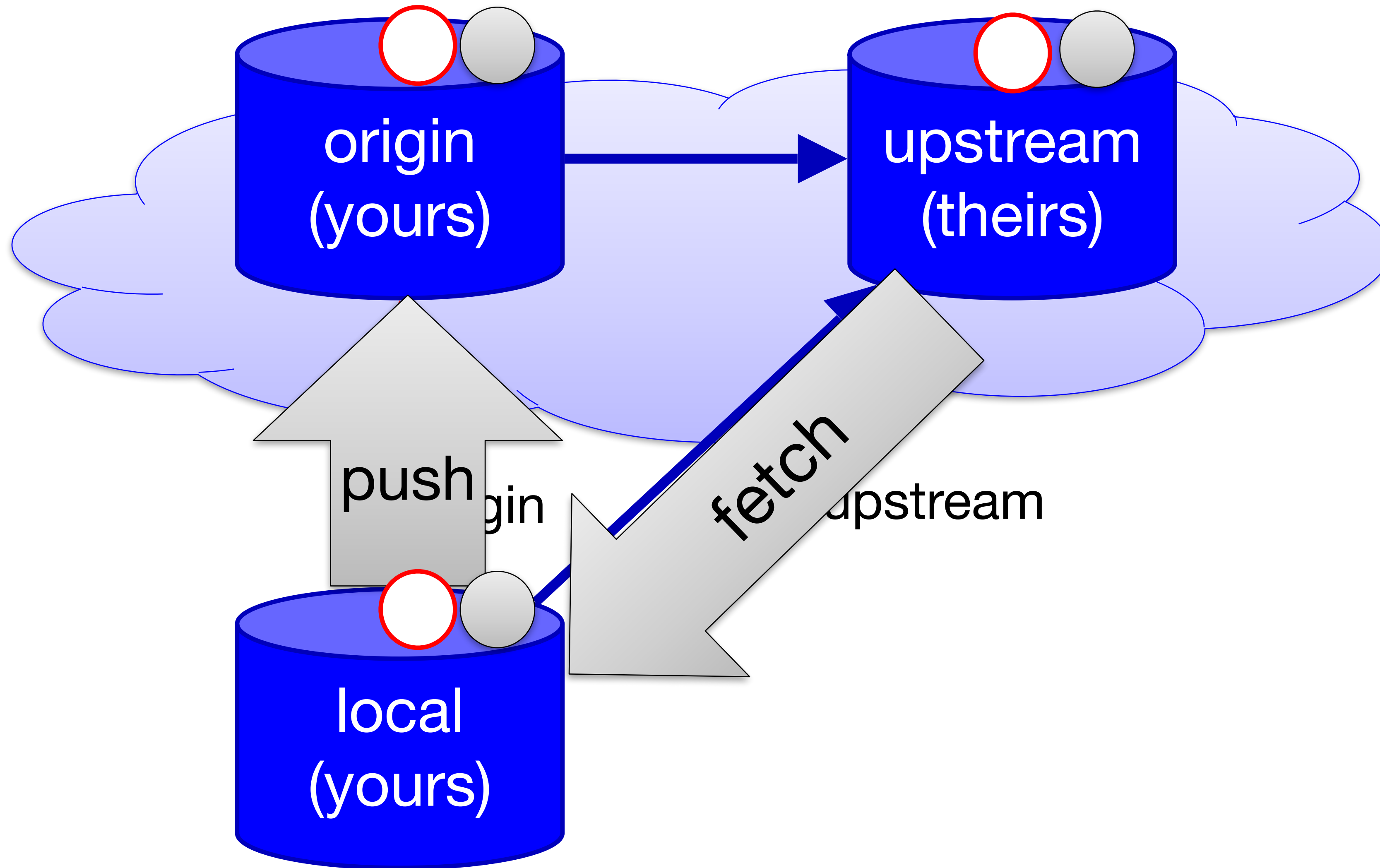
# Integrate Changes



# Integrate Changes



# Integrate Changes

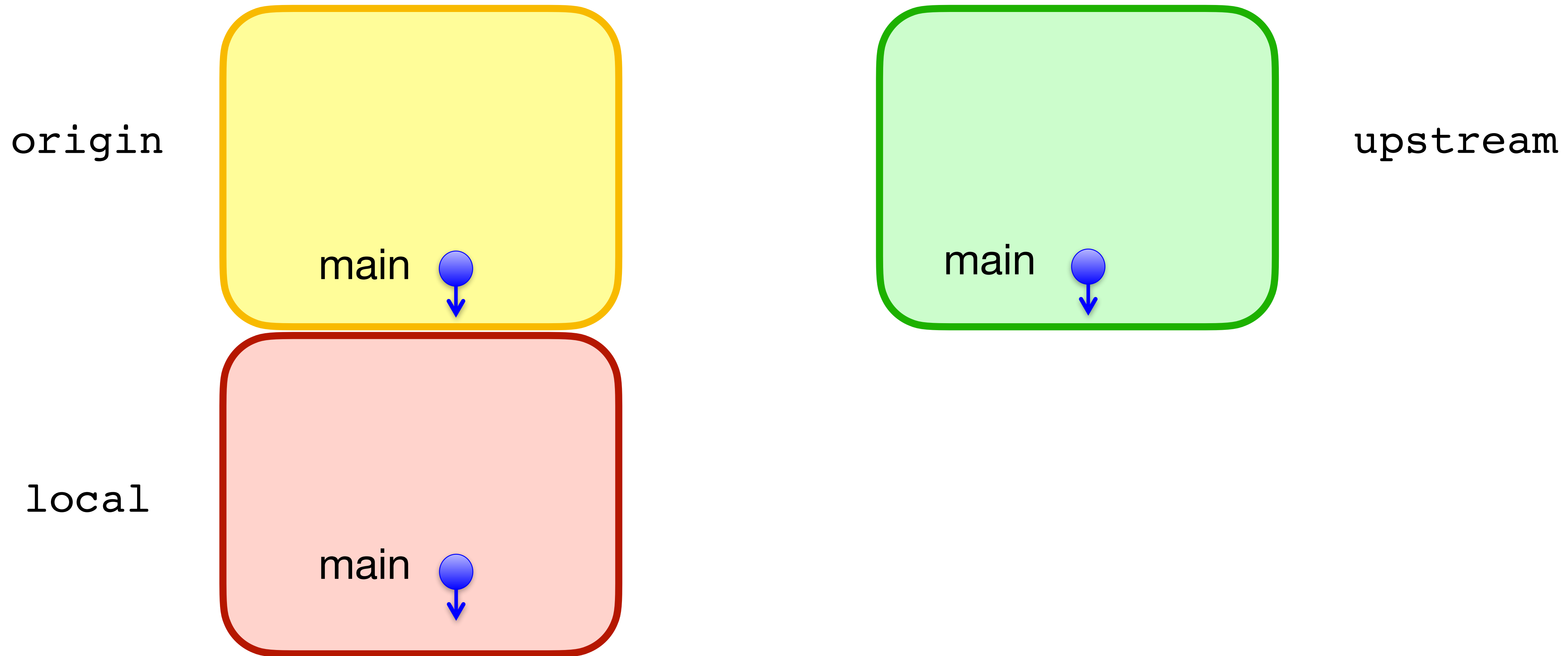


You want to contribute code to the Github project `fancy/project` (`fancy` is the name of the owner, `project` is the name of the repo). You fork the repo (producing `student/project`), commit your changes, and push to `student/project`. Next, you make a pull request for `fancy/project`.

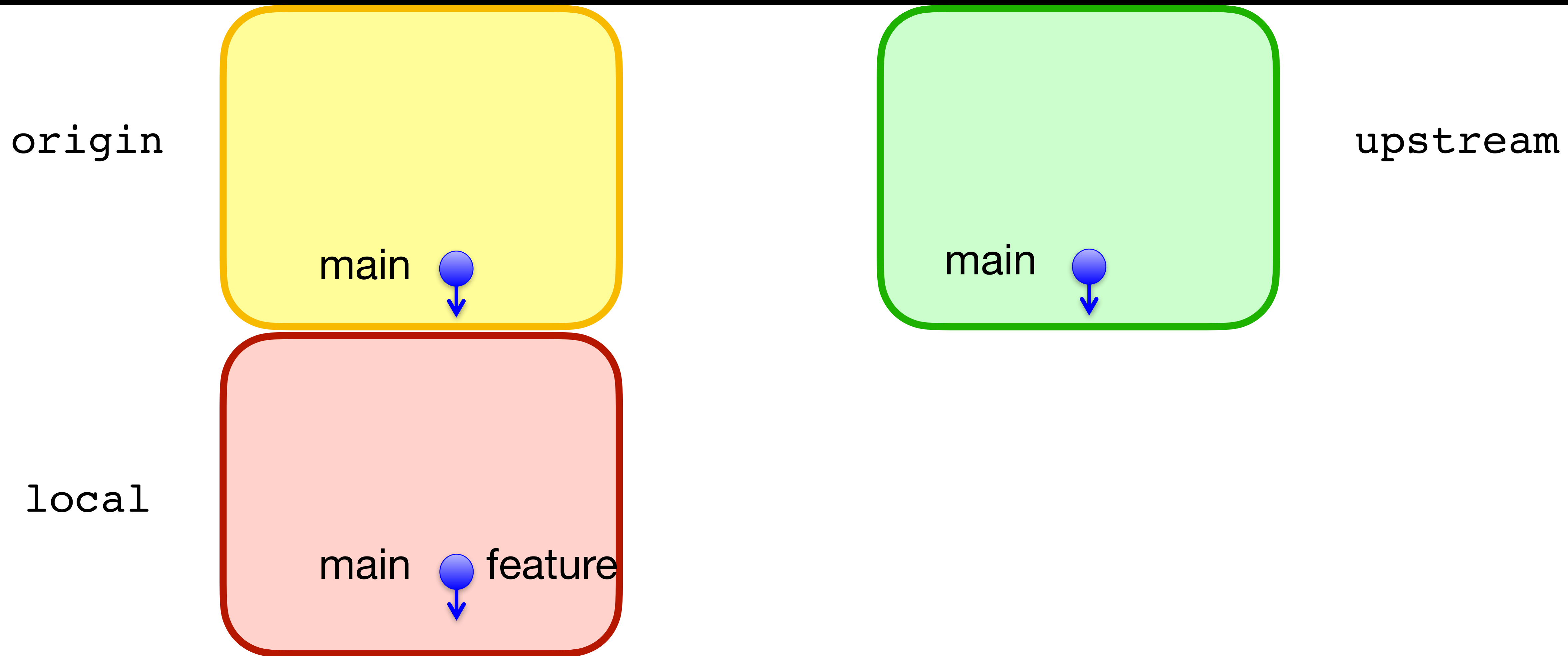
Which statement is true?

- A. Your code is now integrated into `fancy/project` via merging
- B. Your code is now integrated into `fancy/project` via rebasing
- C. You have requested that your code be integrated into `fancy/project`, but no changes have been made
- D. You cannot make any additional commits until the pull request has been accepted

# Branches

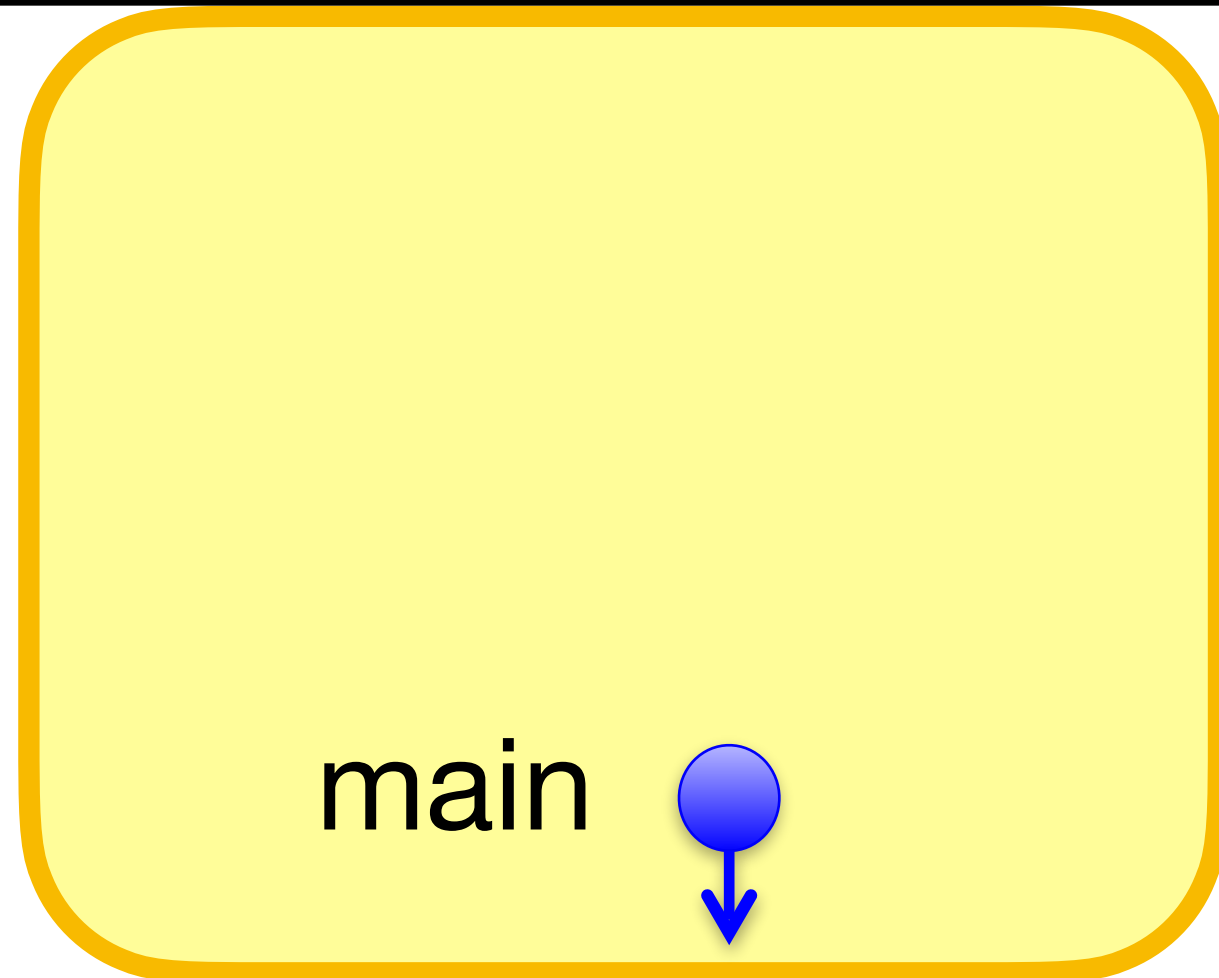


```
$ git checkout -b feature
```

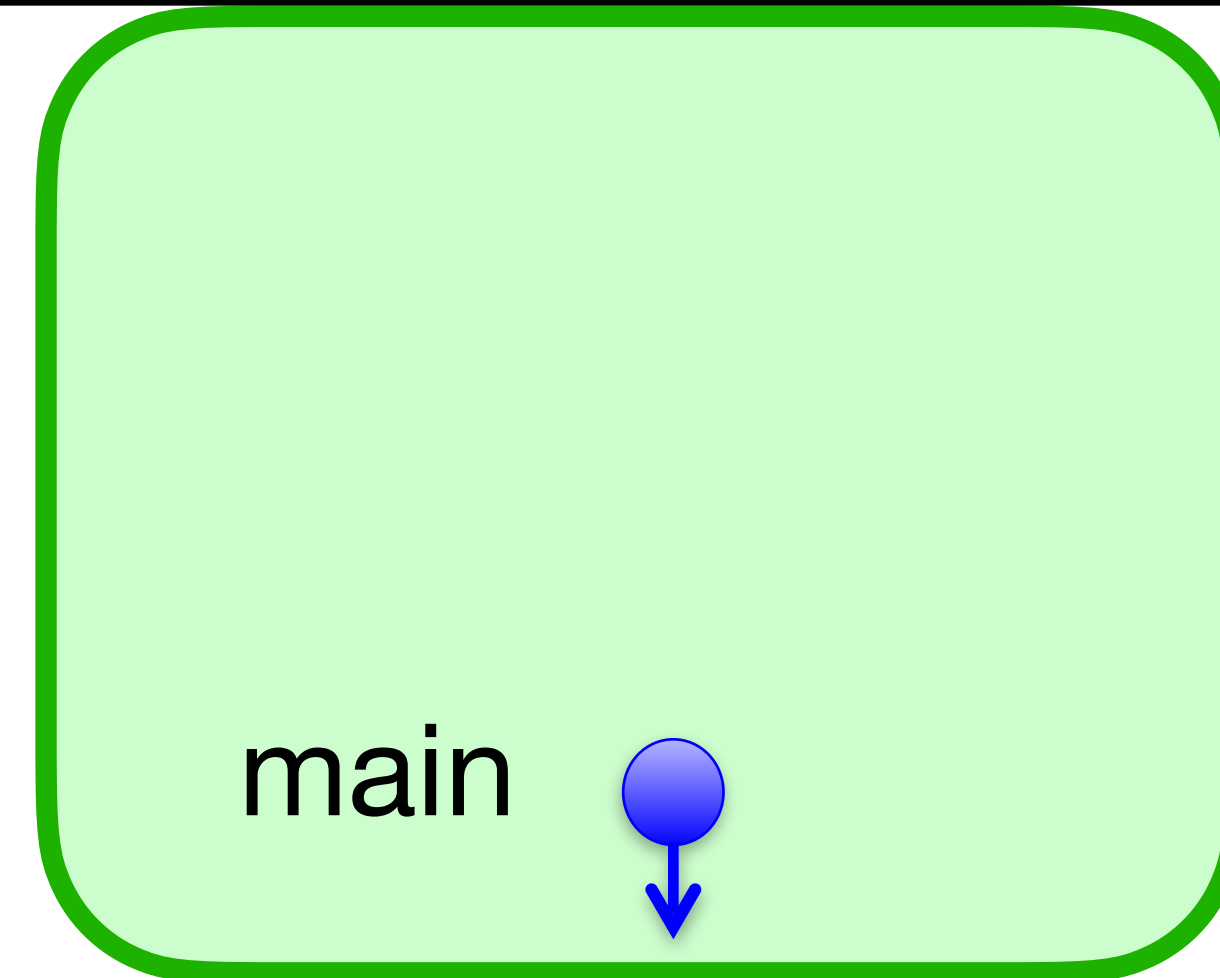


```
$ git commit
```

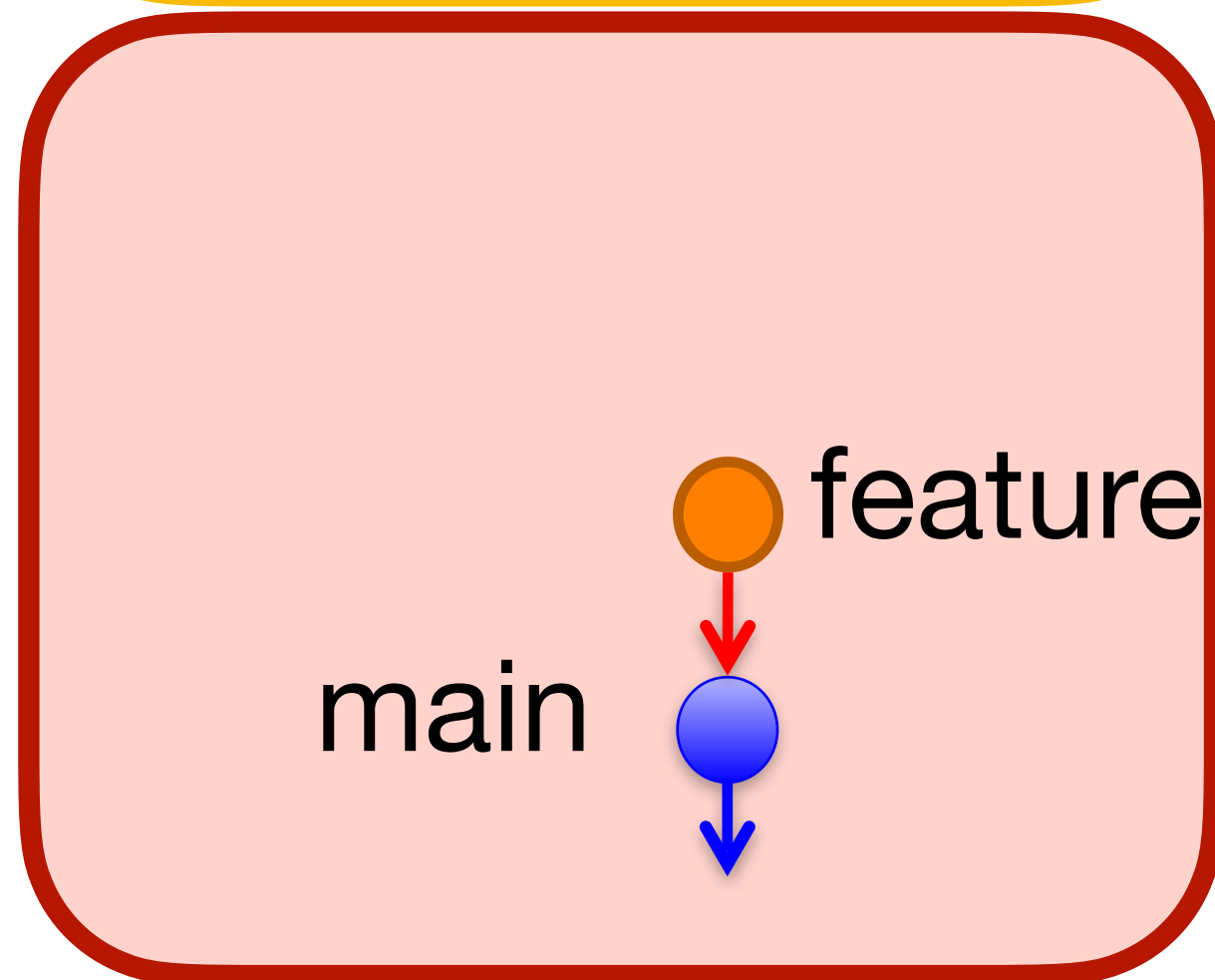
origin



upstream



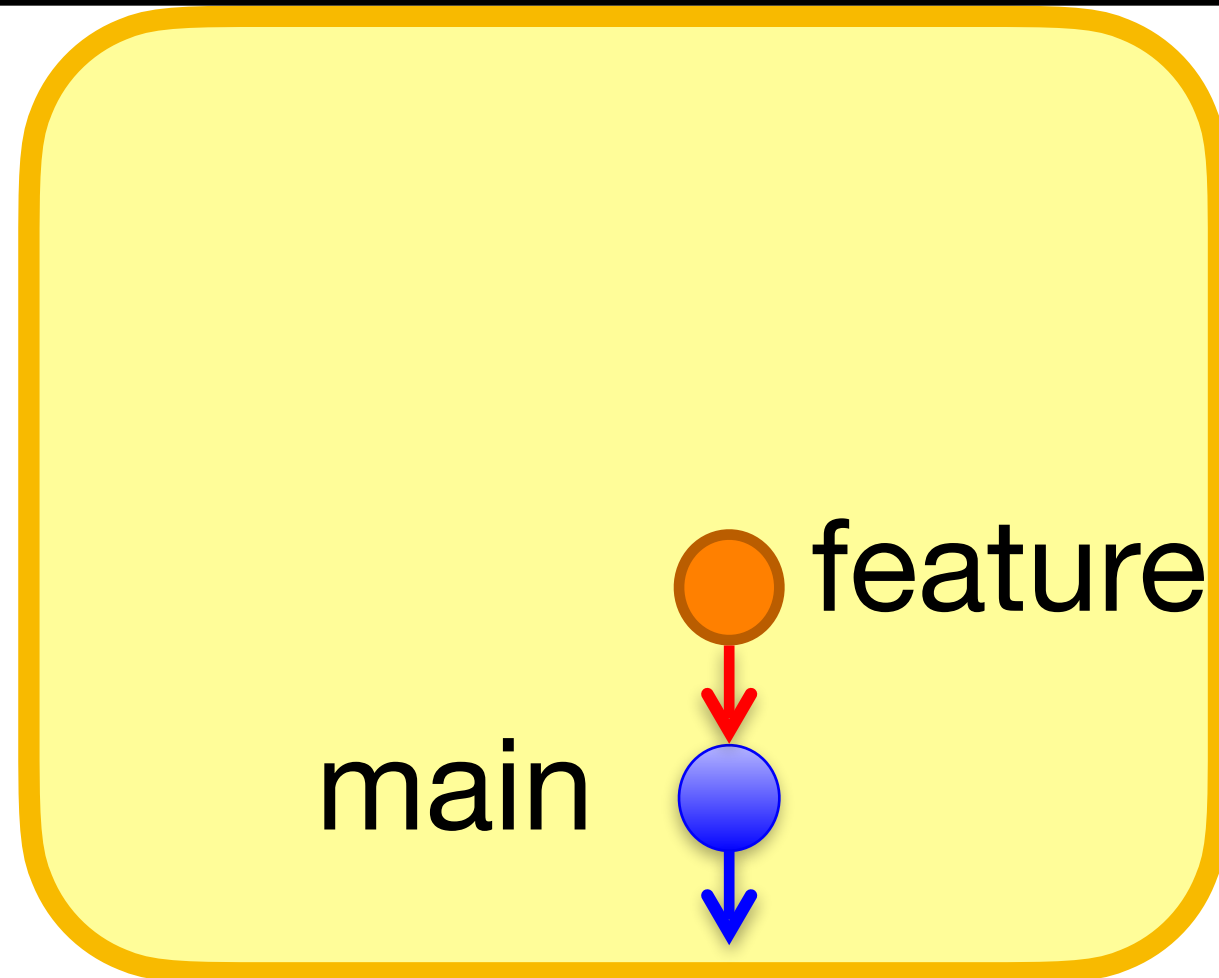
local



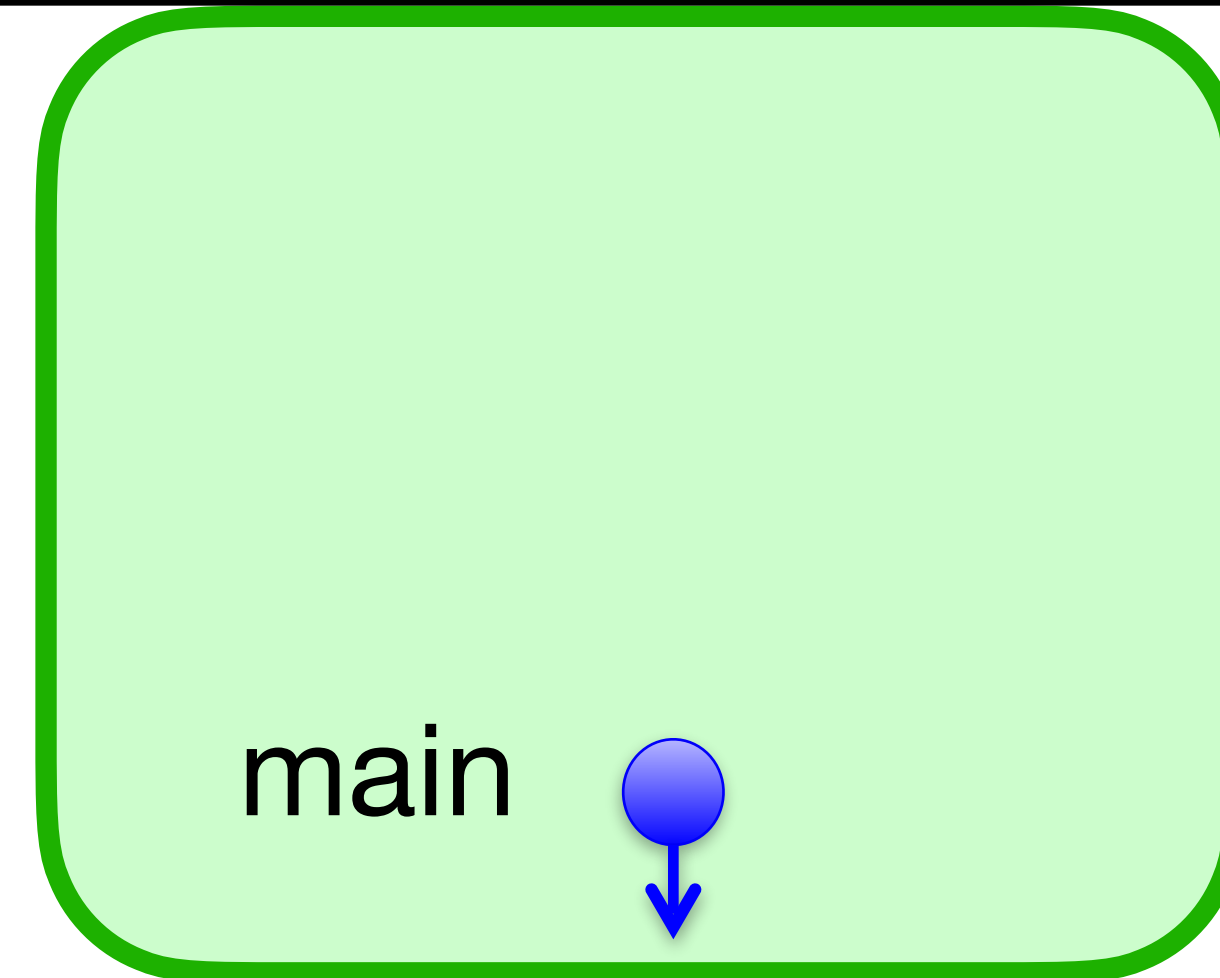


```
$ git push -u origin feature
```

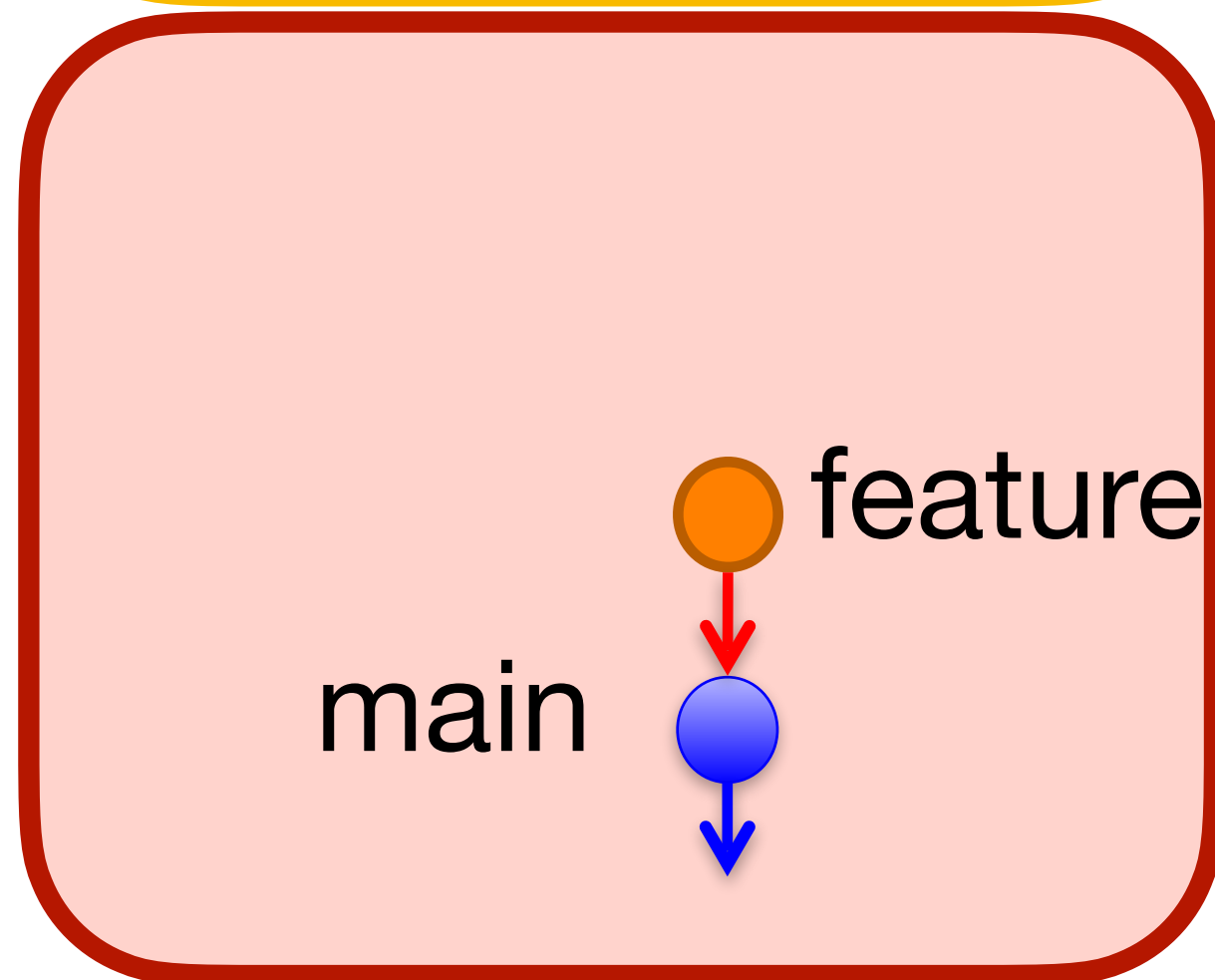
origin



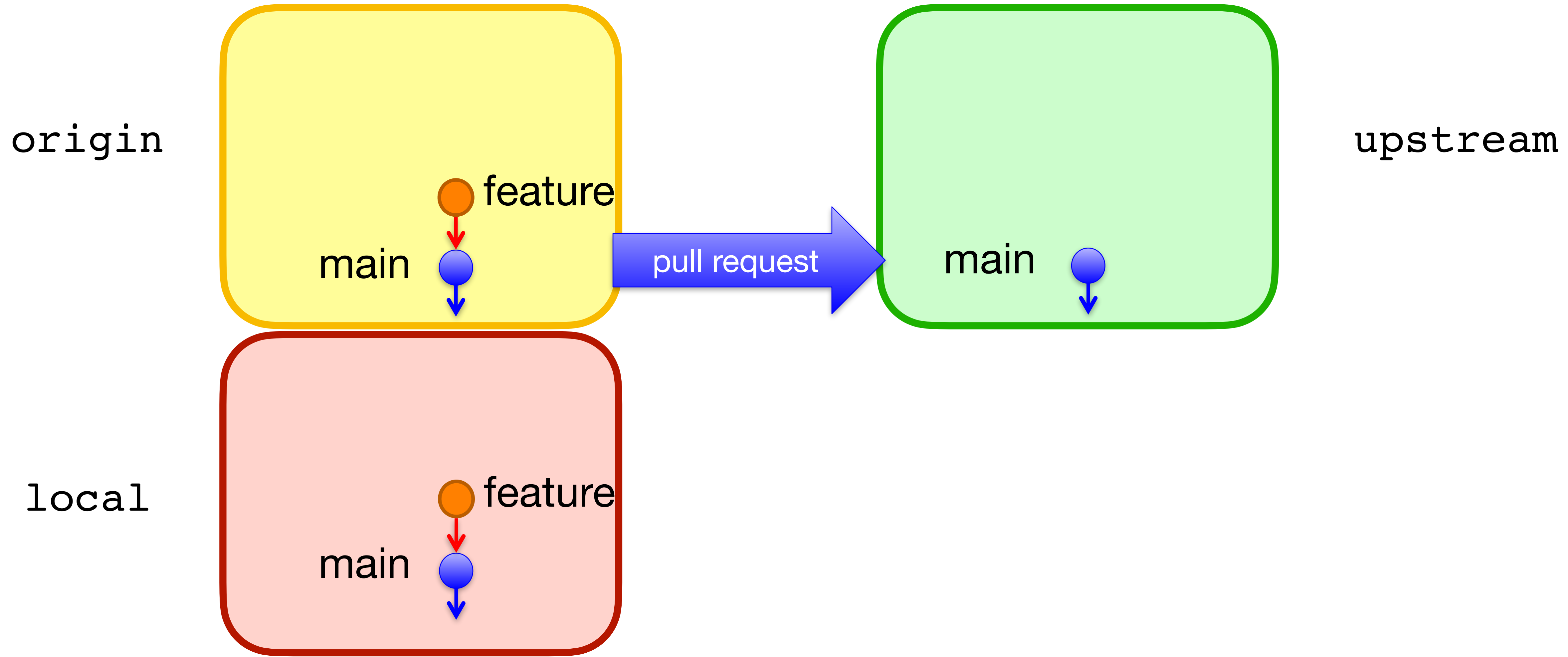
upstream



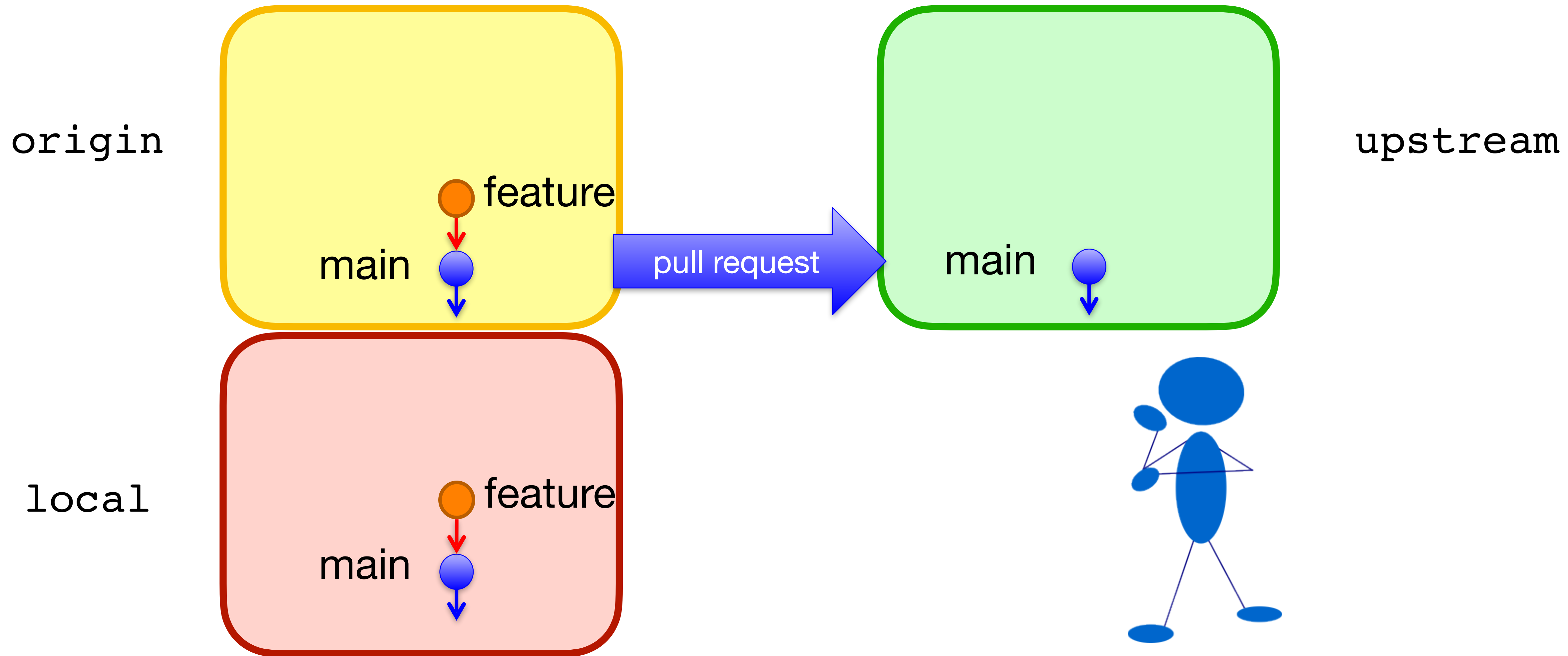
local



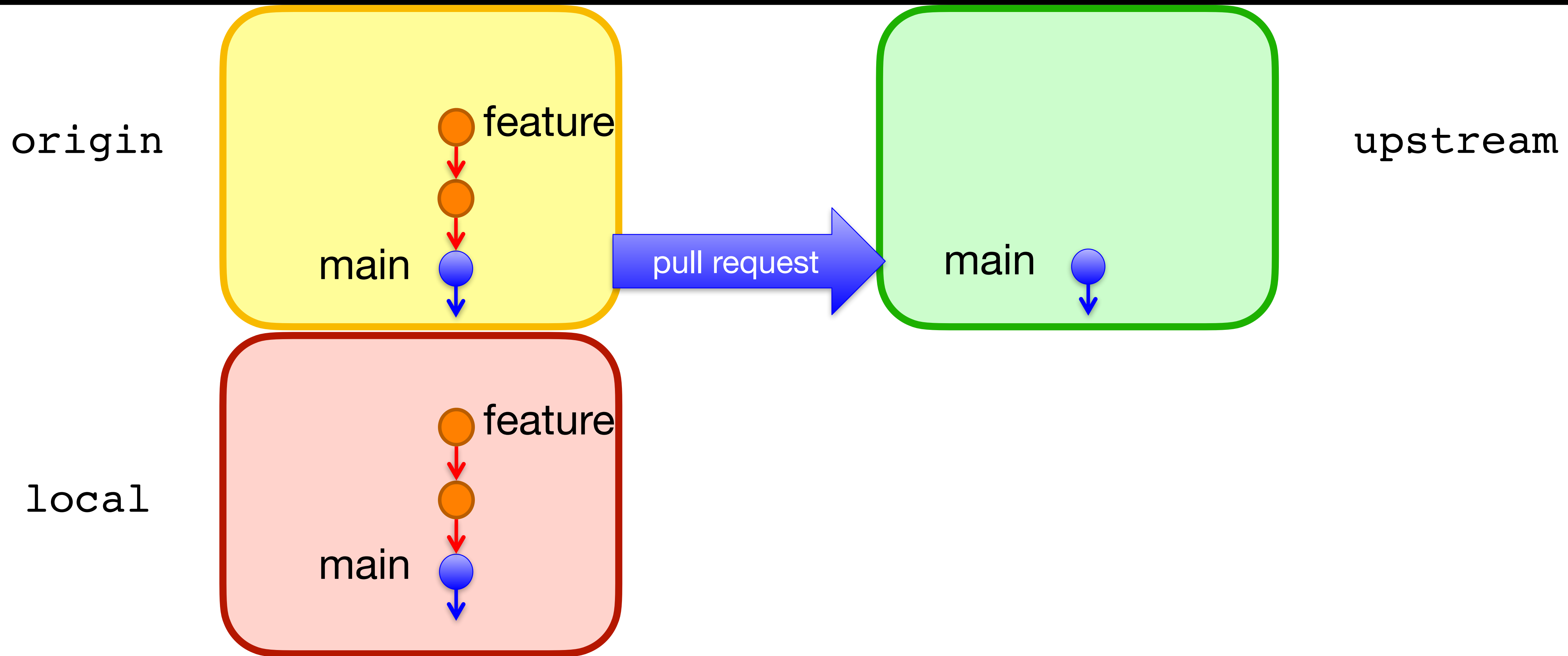
# New pull request



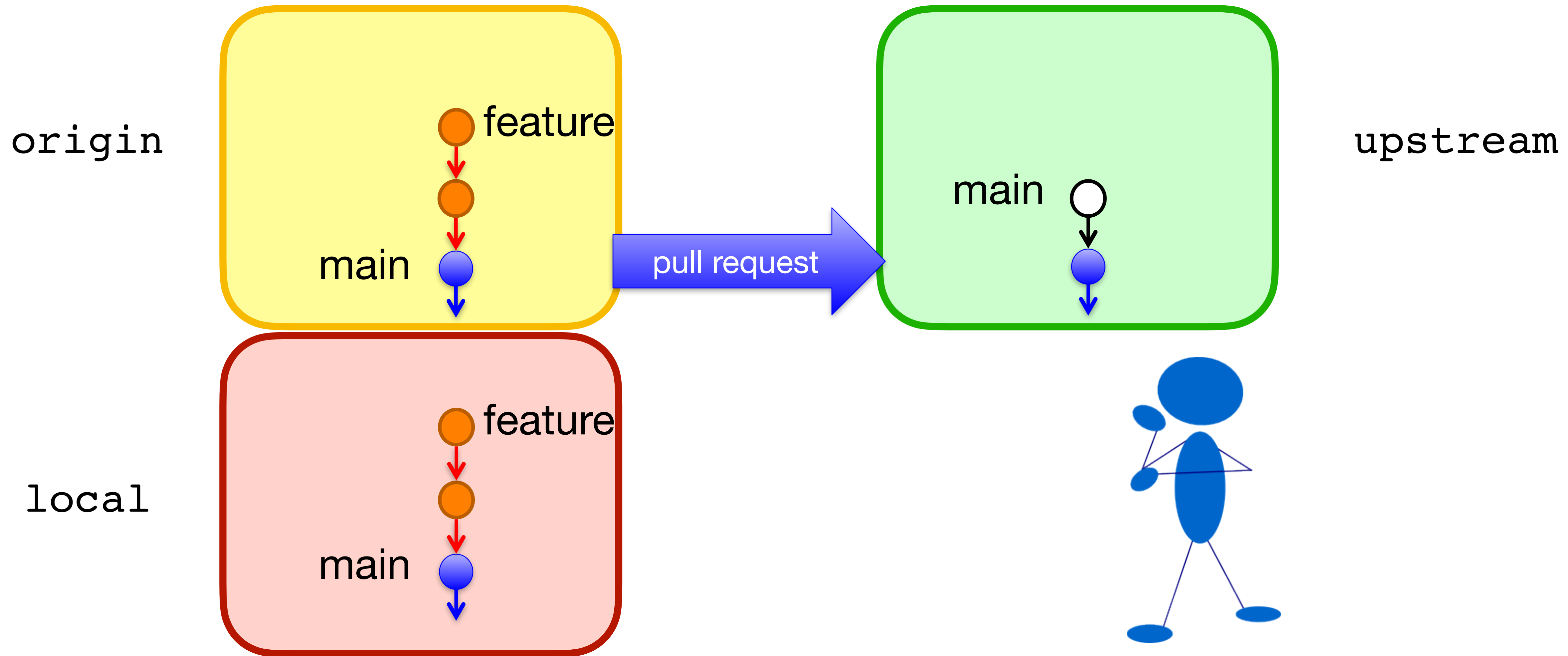
# Great idea, now can you do it more like this?



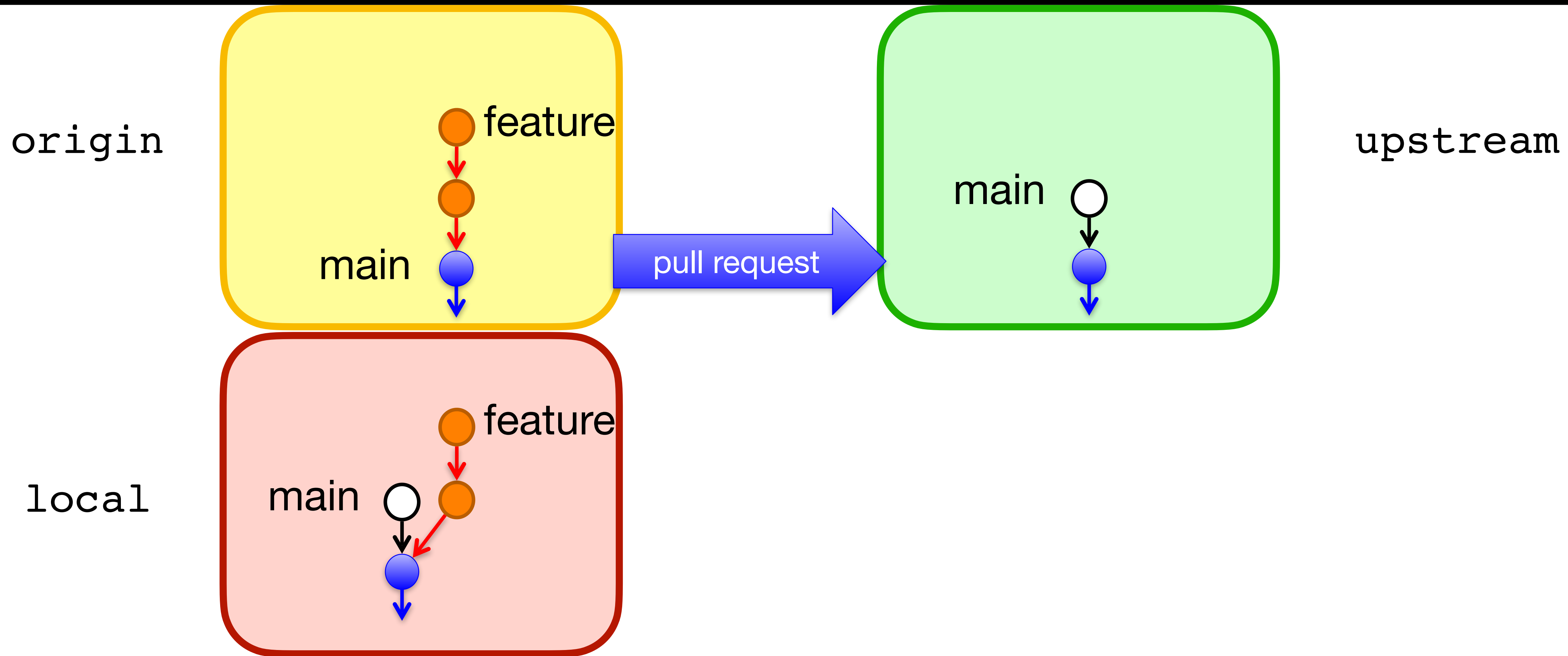
```
$ git commit  
$ git push
```



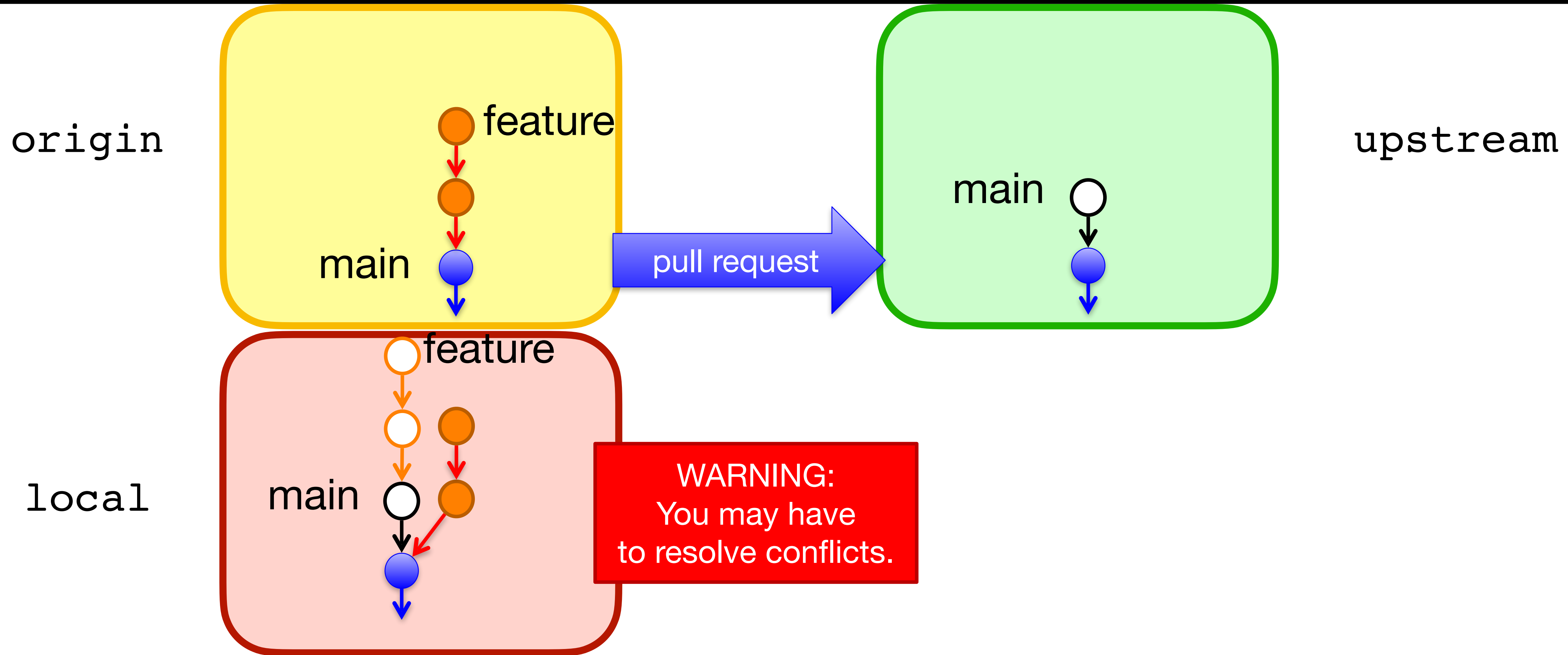
# Awesome, but please update with new changes in main



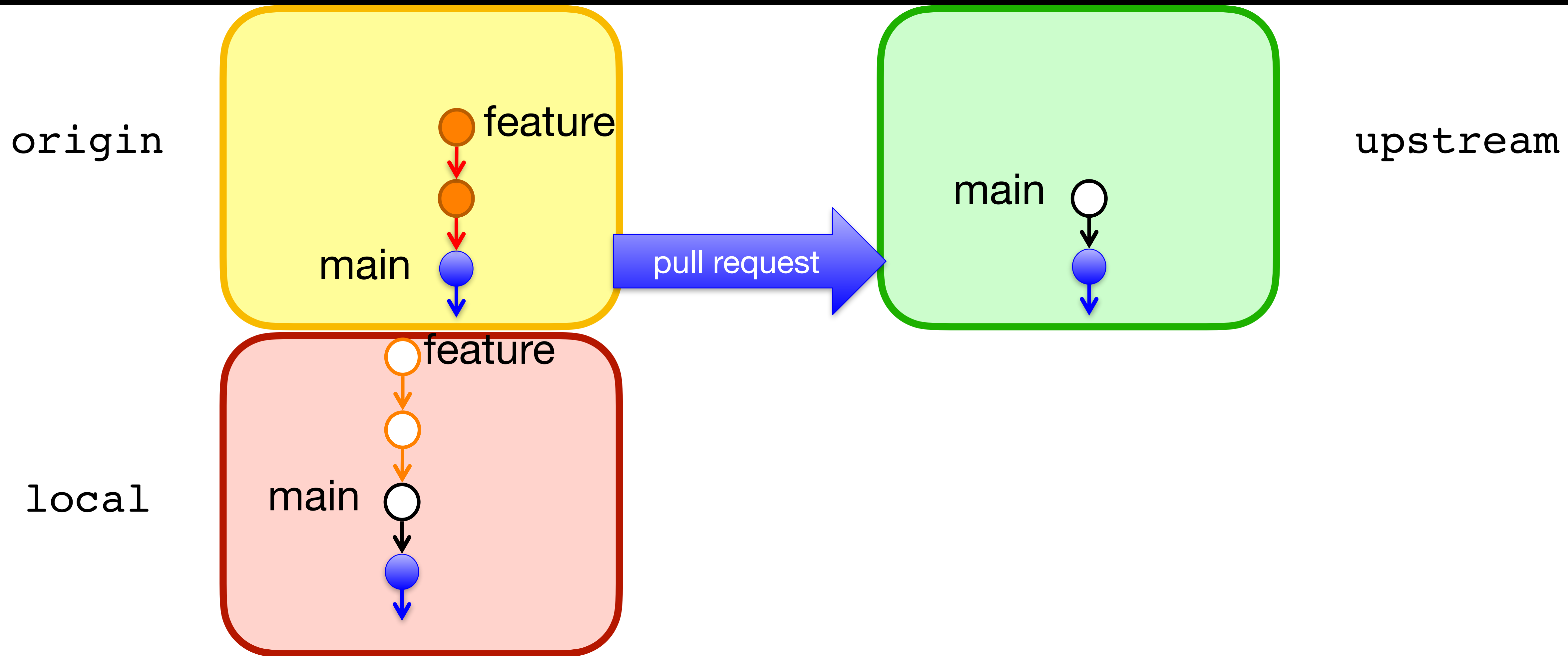
```
$ git remote add upstream https://github.com/...  
$ git fetch upstream main:main
```



```
$ git rebase main
```

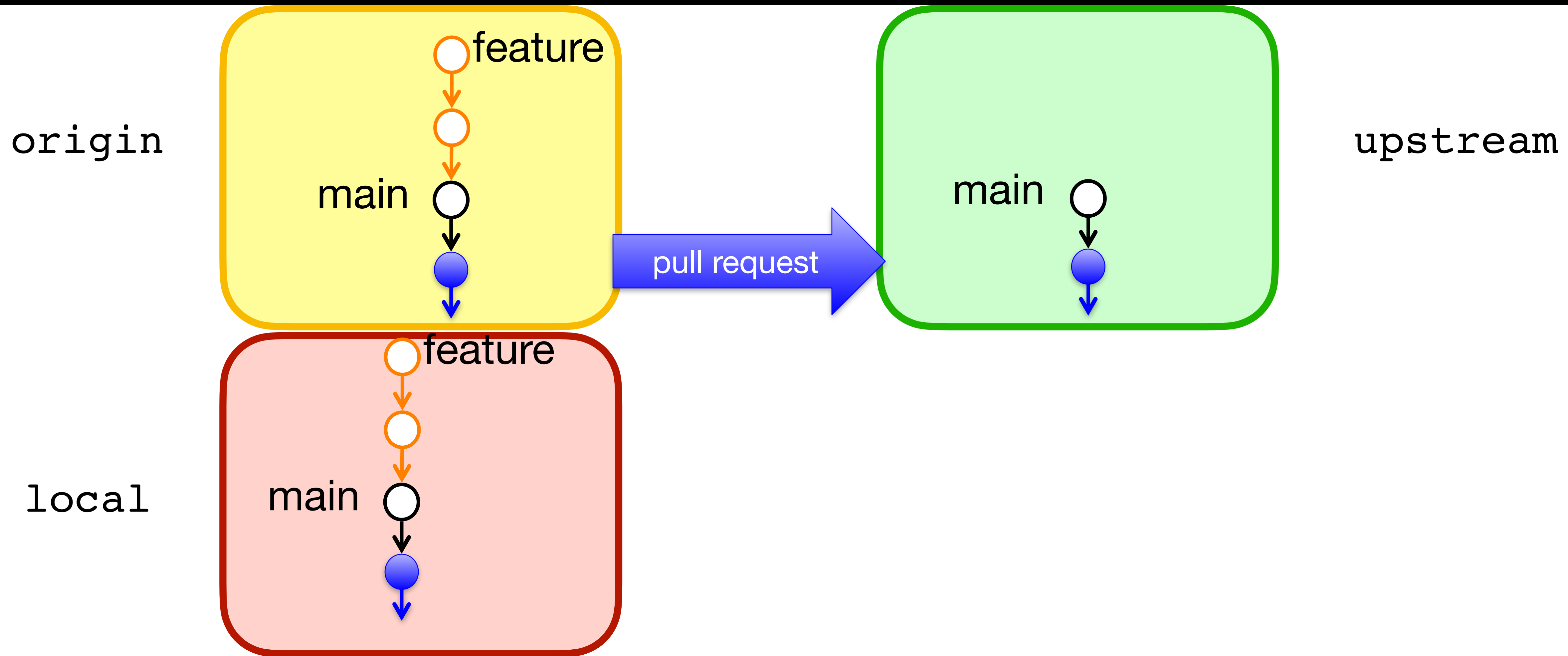


\$ git rebase main

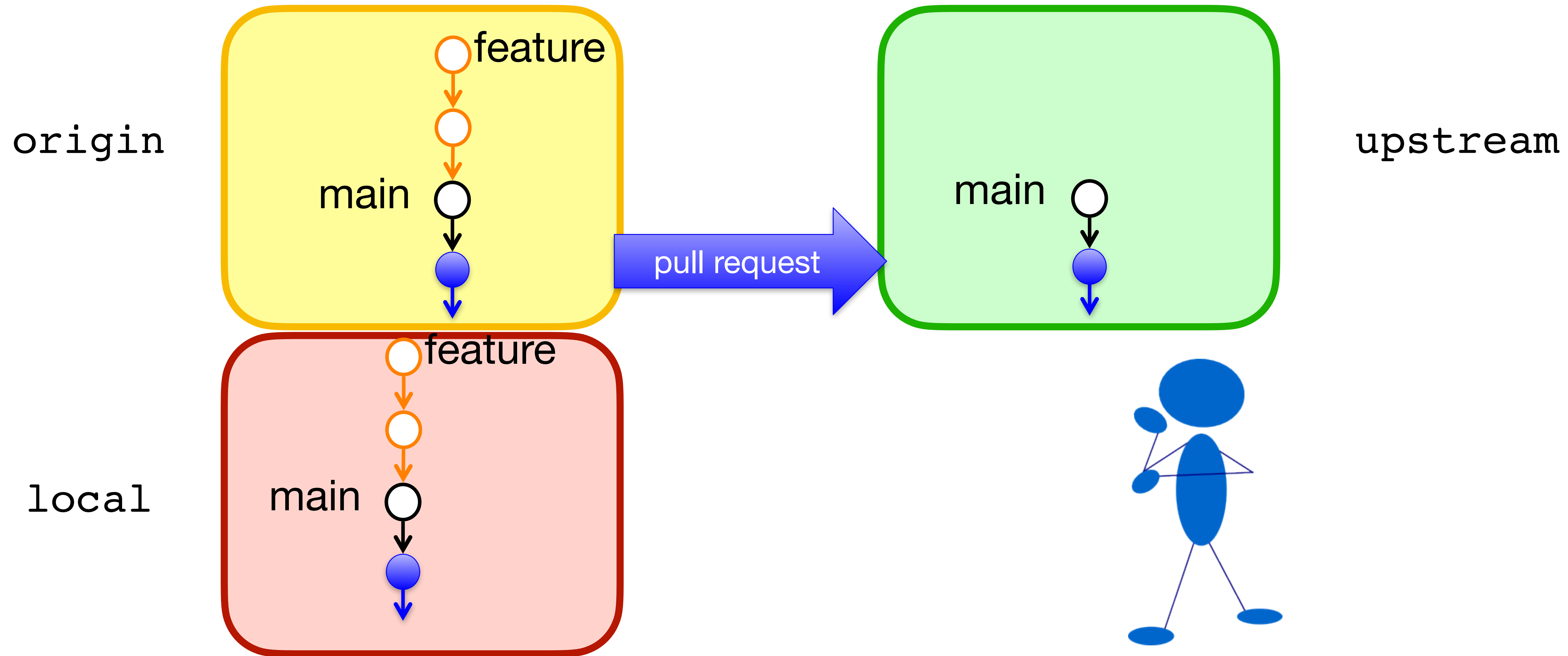




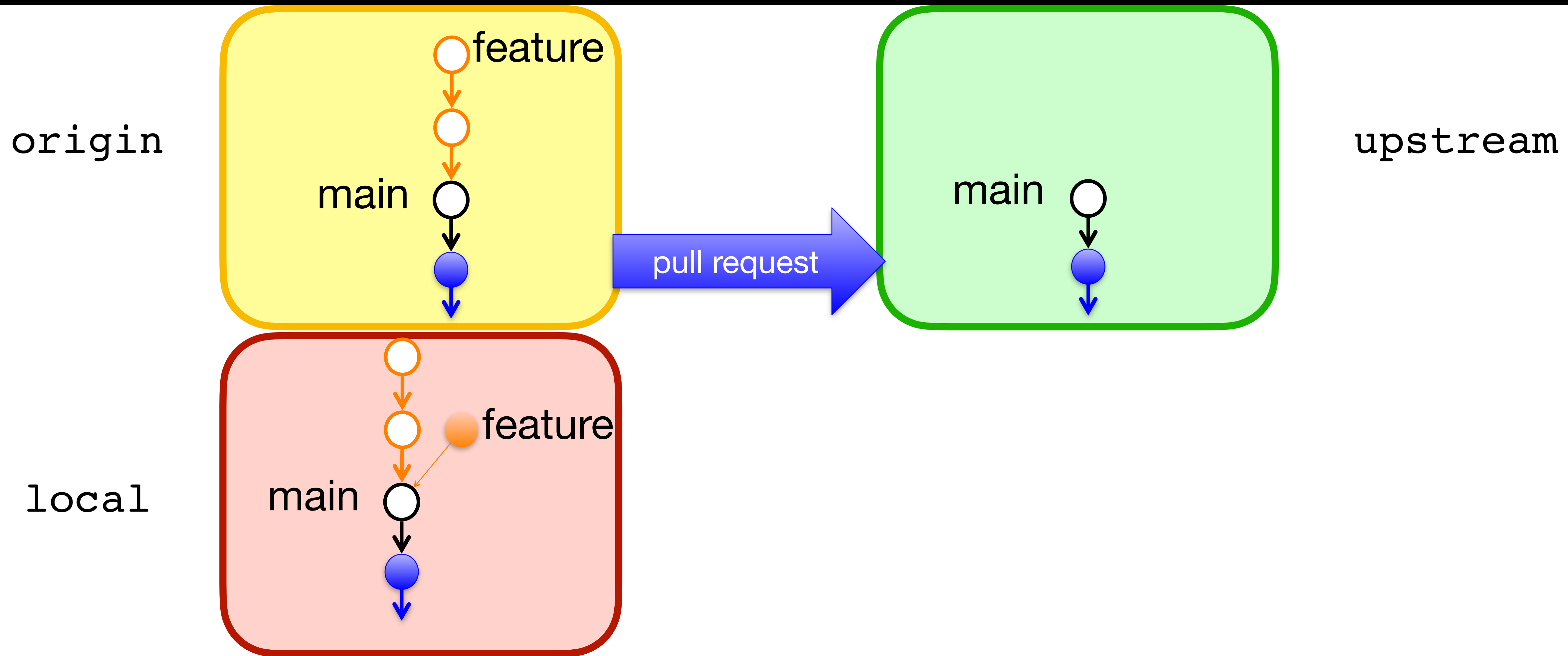
```
$ git push -f origin main feature
```



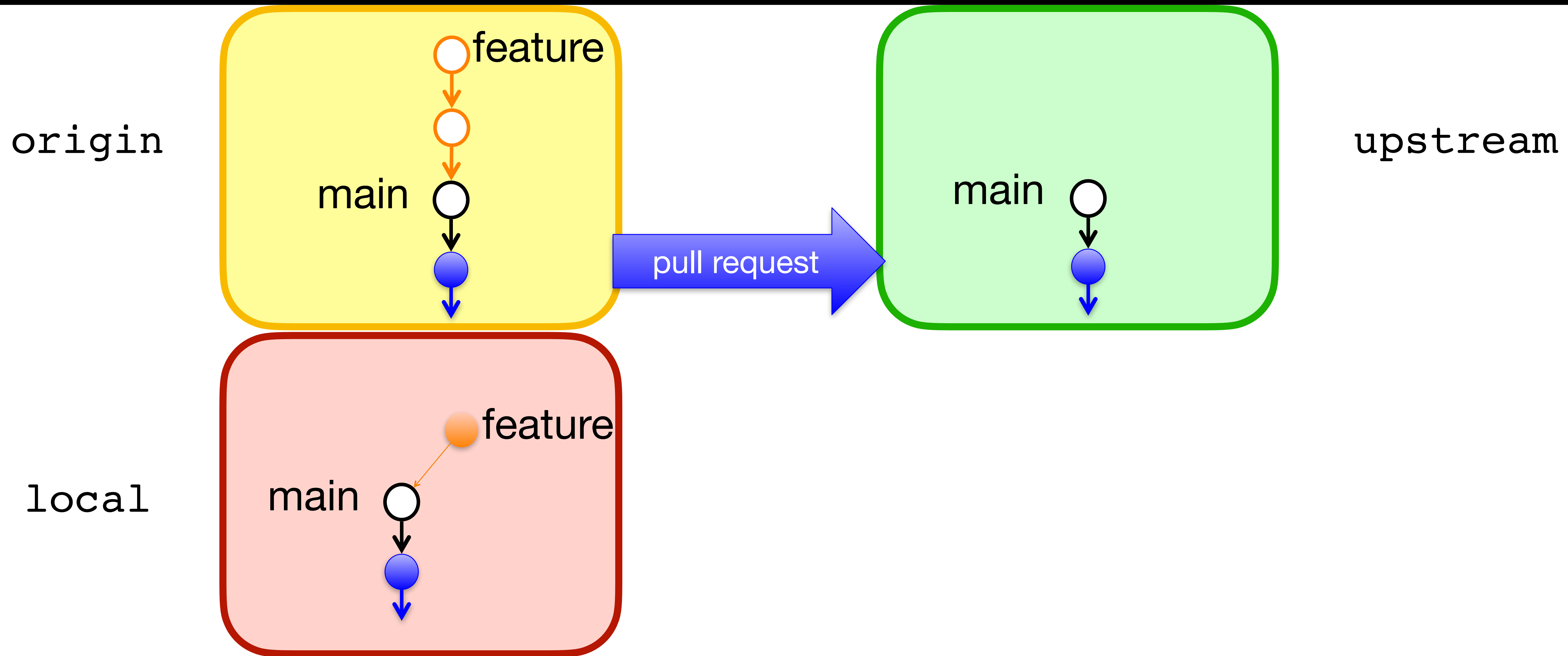
# Great. Please squash your commits.



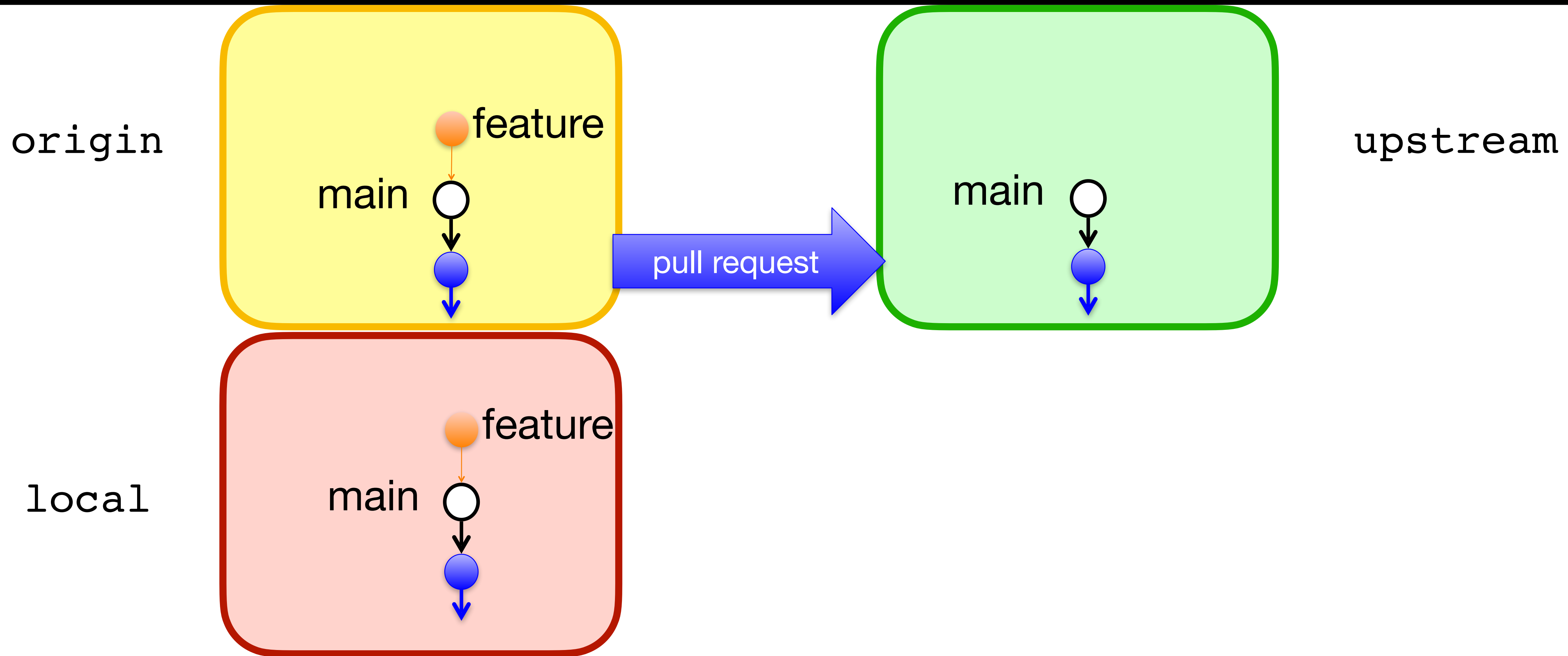
```
$ git rebase -i main
```



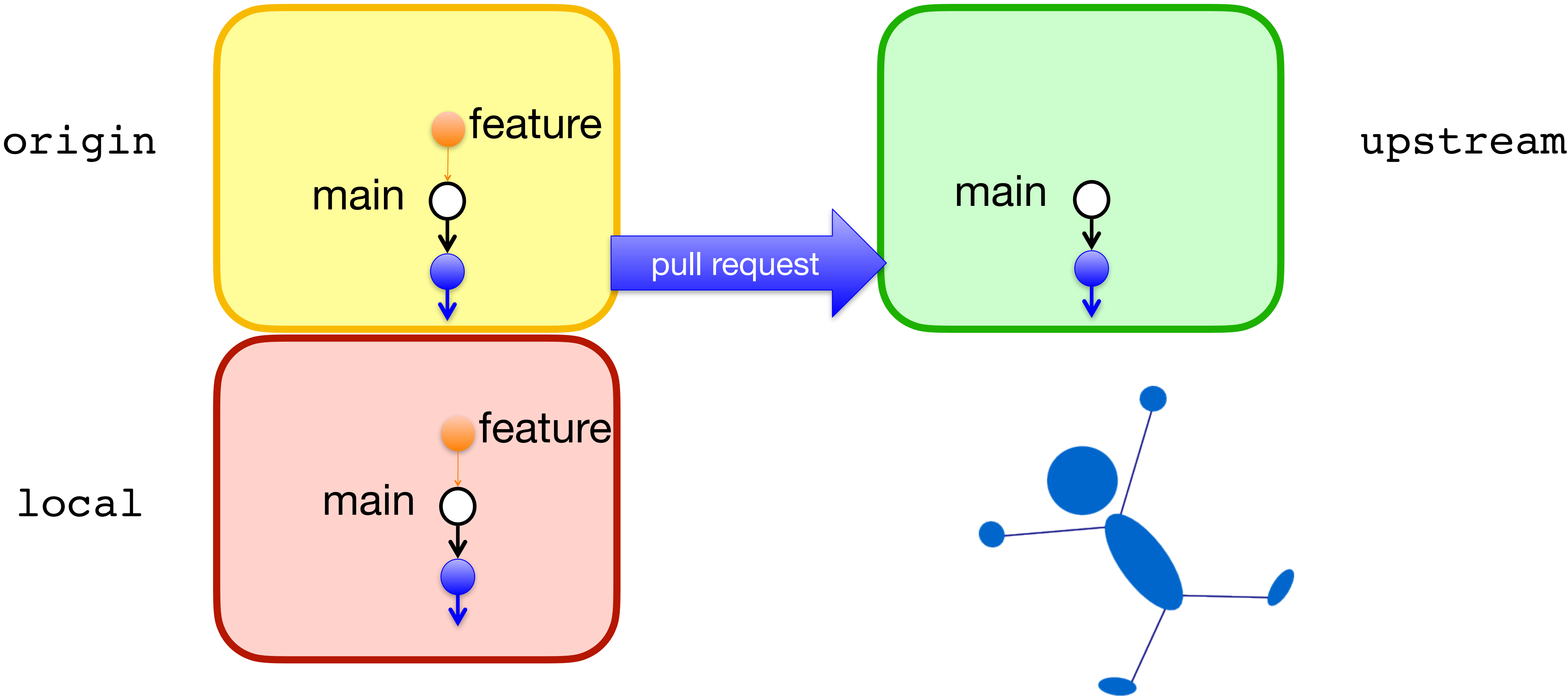
```
$ git rebase -i main
```



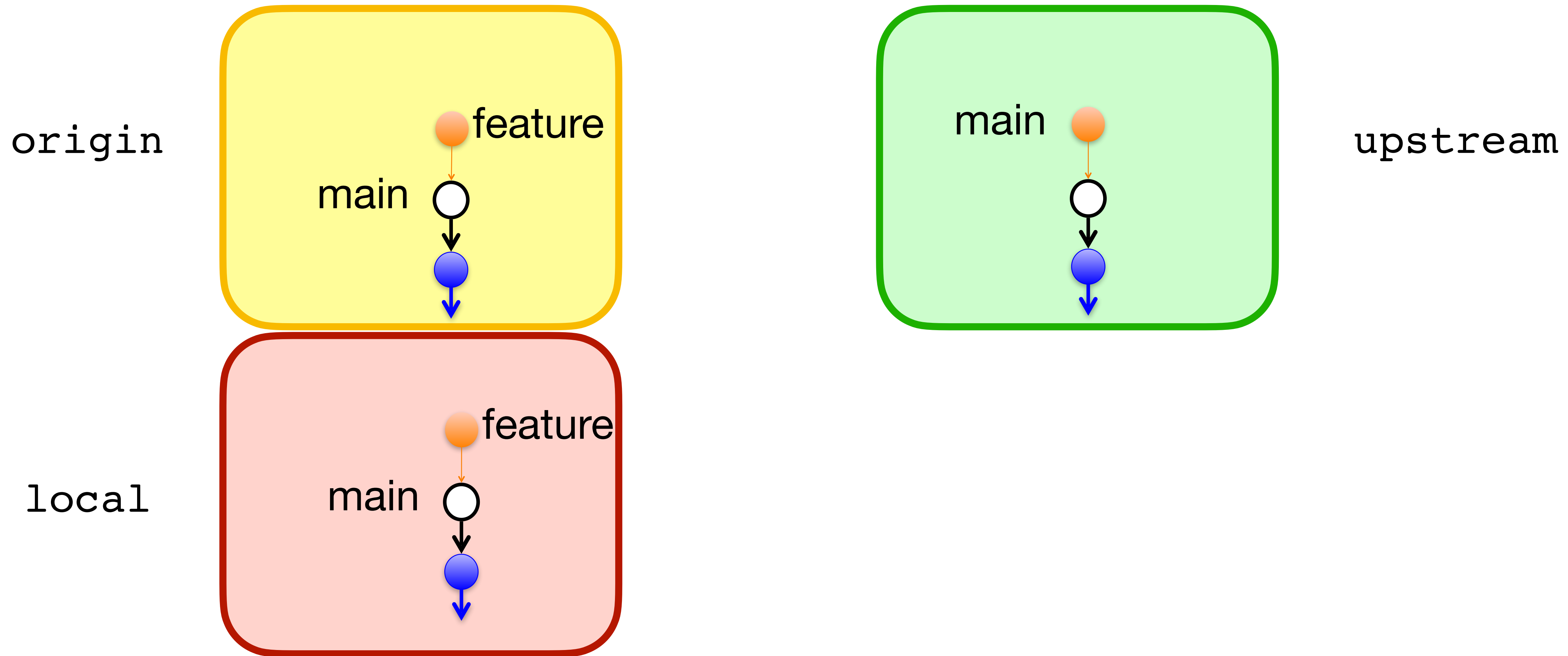
```
$ git push -f origin feature
```



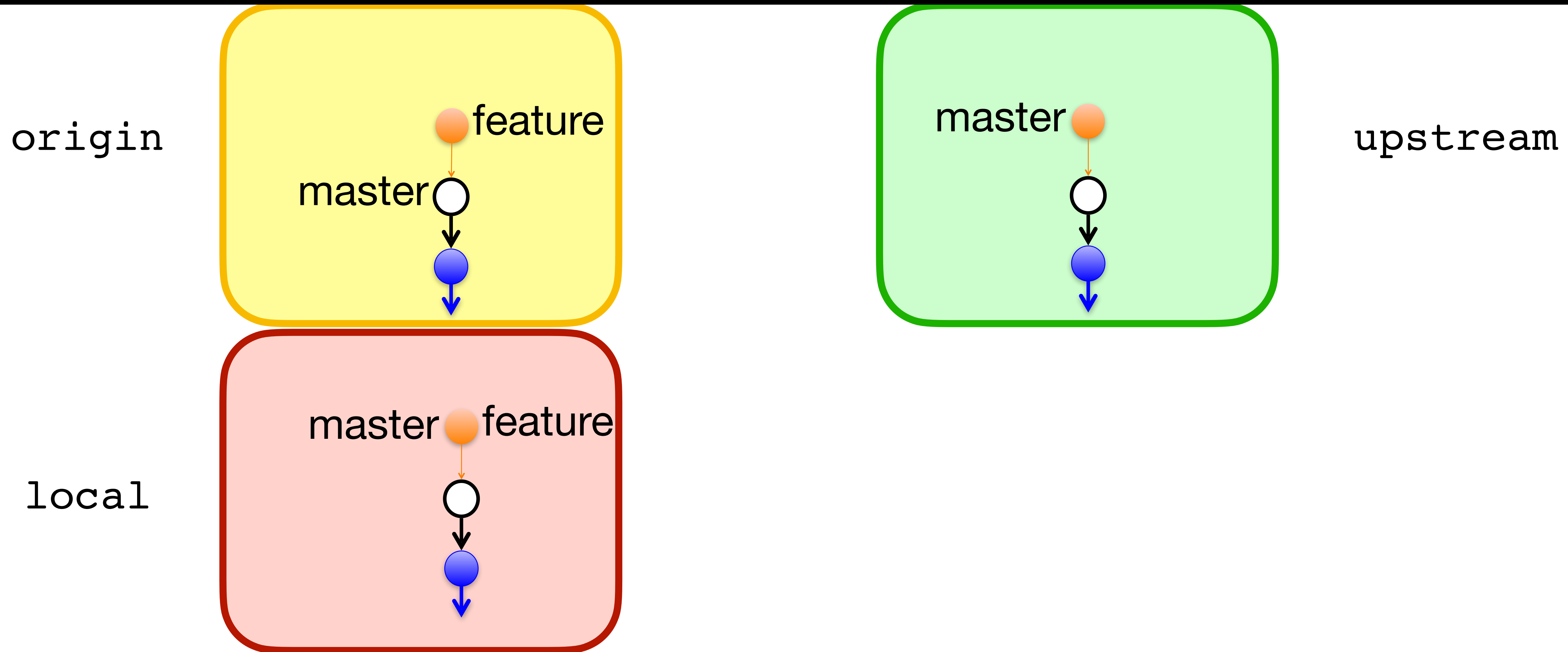
# Perfect, I accept!



# Time to Clean Up

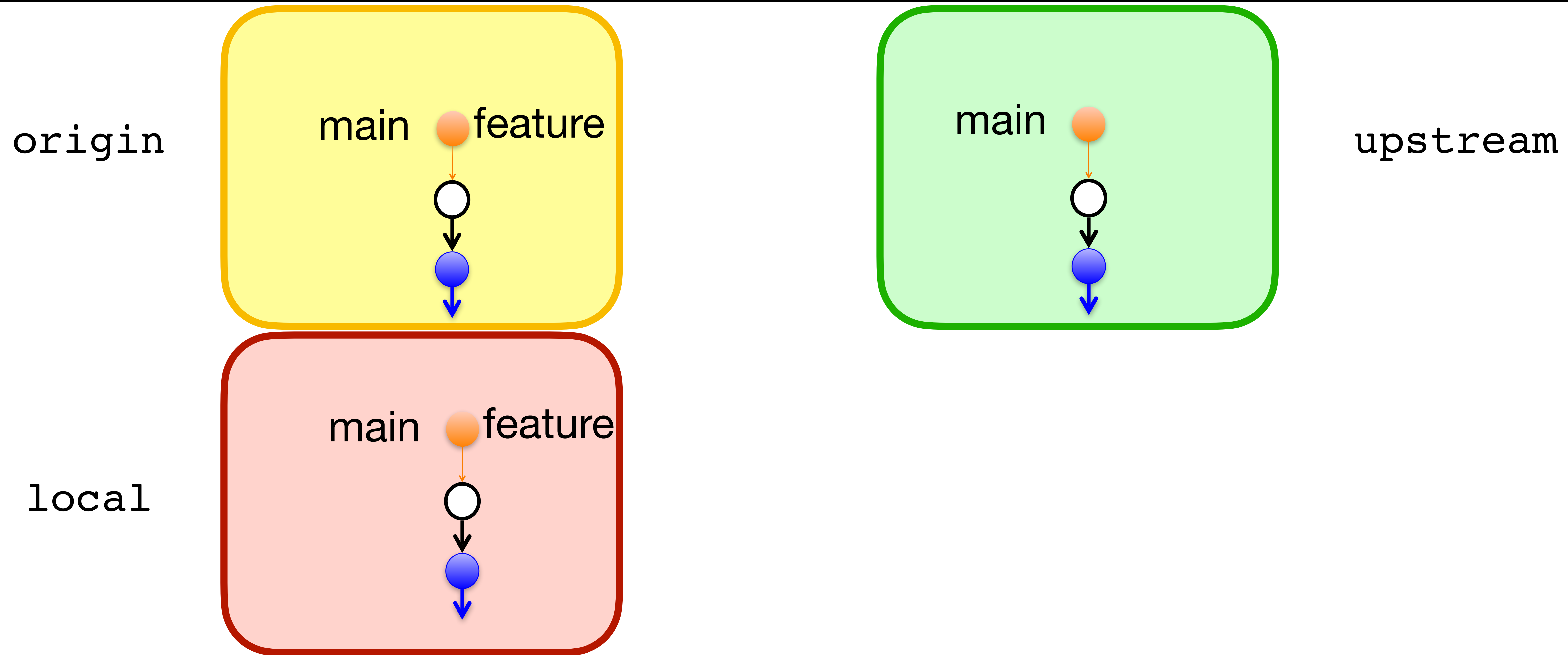


```
$ git fetch upstream main:main
```

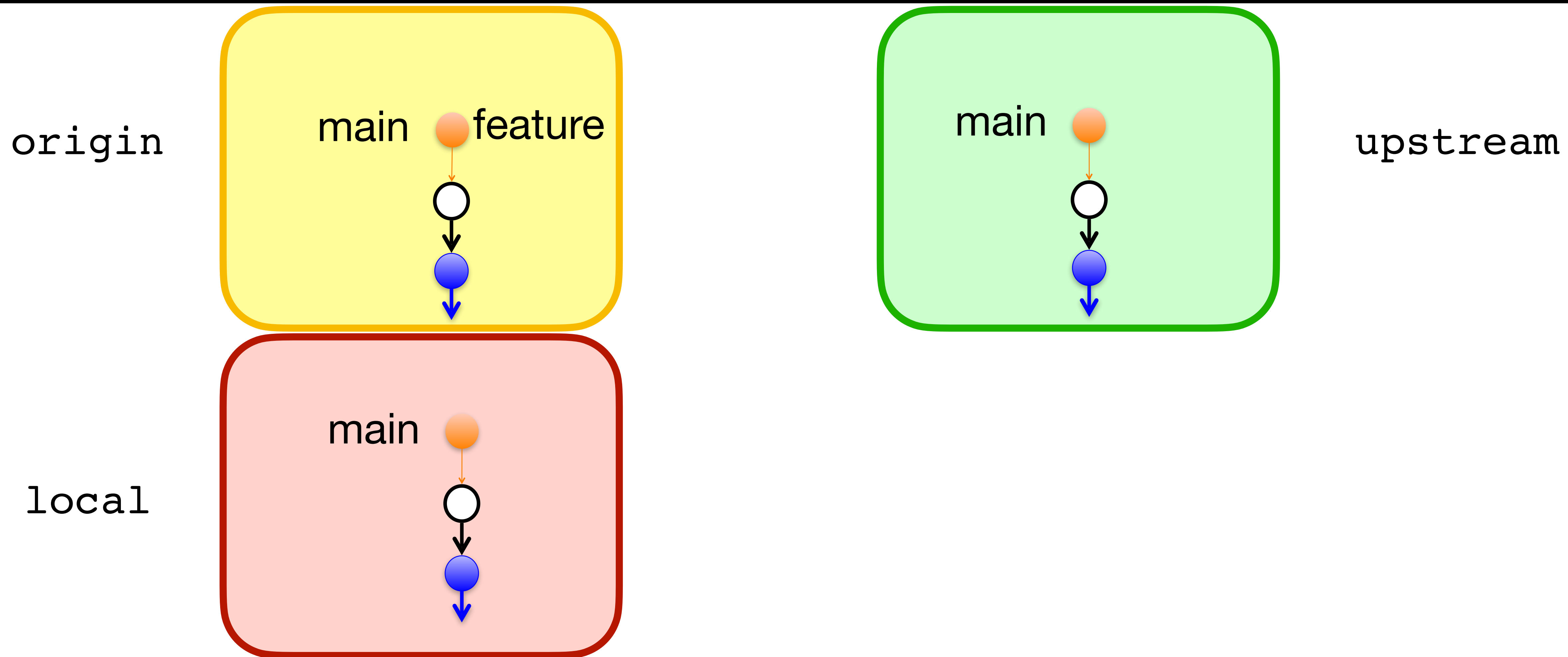




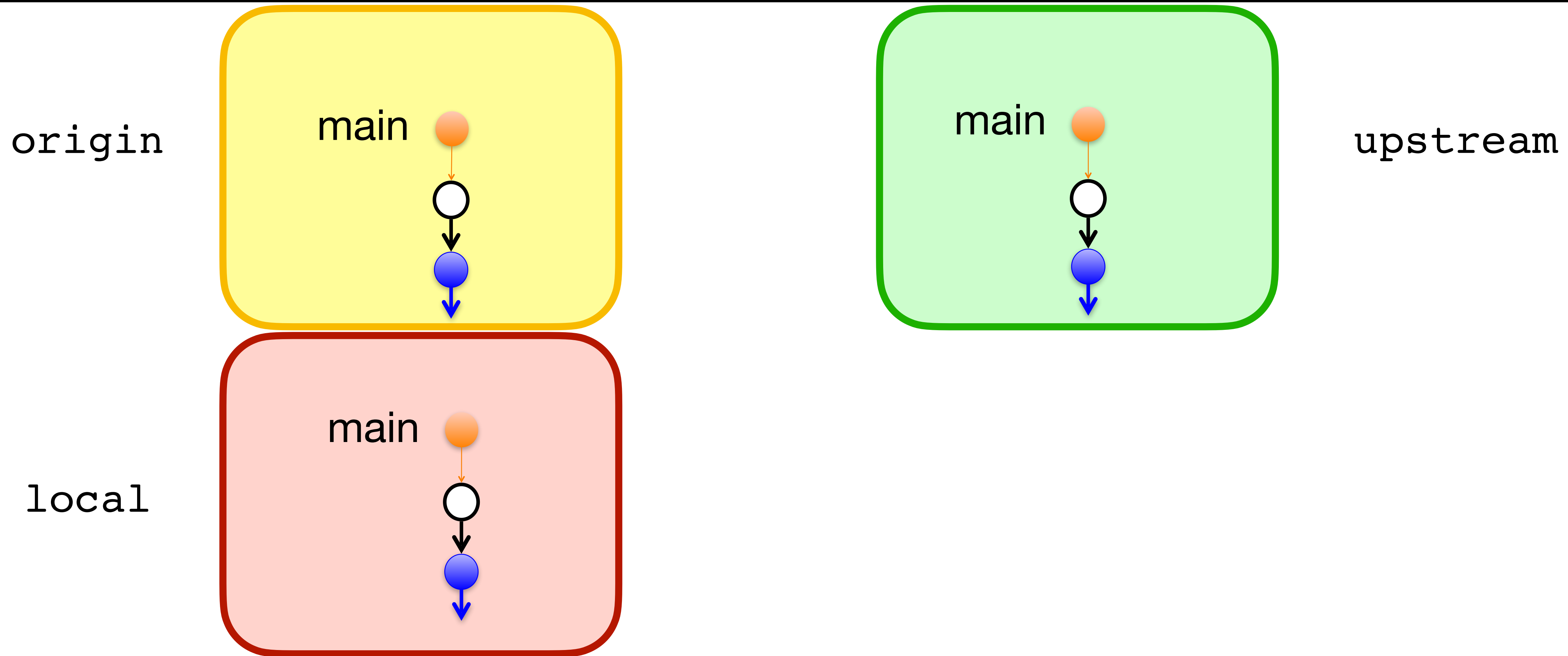
```
$ git push origin main
```



```
$ git checkout main
$ git branch -d feature
```



```
$ git push origin -d feature
```



After a PR is accepted, Github will ask you if you want to delete your feature branch. If you say yes, which branches get deleted?

- A. `feature` — the branch named `feature` in your local repo
- B. `origin/feature` — the branch named `feature` in your remote repo
- C. `upstream/feature` — the branch named `feature` in their remote repo
- D. `feature` and `origin/feature`
- E. `feature`, `origin/feature`, and `upstream/feature`

Now that `origin/feature` has been deleted, how do you delete feature?

A. `$ git delete feature`

B. `$ git delete -b feature`

C. `$ git branch -d feature`

D. `$ git push origin -d feature`

E. I would google "delete a git branch" and then click on <https://stackoverflow.com/questions/2003505/how-do-i-delete-a-git-branch-locally-and-remotely> like every other programmer