From version control to GitLab

EURO-LABS Advanced Training

Open Science and Data Management

Content

- Version Control
  - What is it?
  - How can it help you?
- GIT
  - Under the hood
  - Branching and Merging
- GitLab
  - GUI
  - Workflows

Underlying Problem

- How do you keep track of changes made to your files?
  - Do the bookkeeping yourself.

    Create new files every time you do a change.
    - Will create a huge number of files which are complicated to organize
    - May blow up your disk space
    - No metadata information if not saved separately
- How do you communicate changes to your collaborators?
  - Send changes by mail?
  - Work all in the same account?
- Use a system which hides all the complications from the user

Version Control

- A version control system (VCS) manages documents over time
- A VCS keeps the history of all changes
  - Many versions of every file
  - Allows to go back to an older version of the file
  - Show differences between different versions
  - Log messages name the reason for changes
  - ...
- A VCS coordinates the work of multiple authors
  - Avoid conflicts between the developments of different developers
- A VCS allows user authentication and controlled access to files
  - Read/Write permissions for user and groups
  - Permissions can apply for repo, directory, or file (depend on VCS)

Which Information is Stored

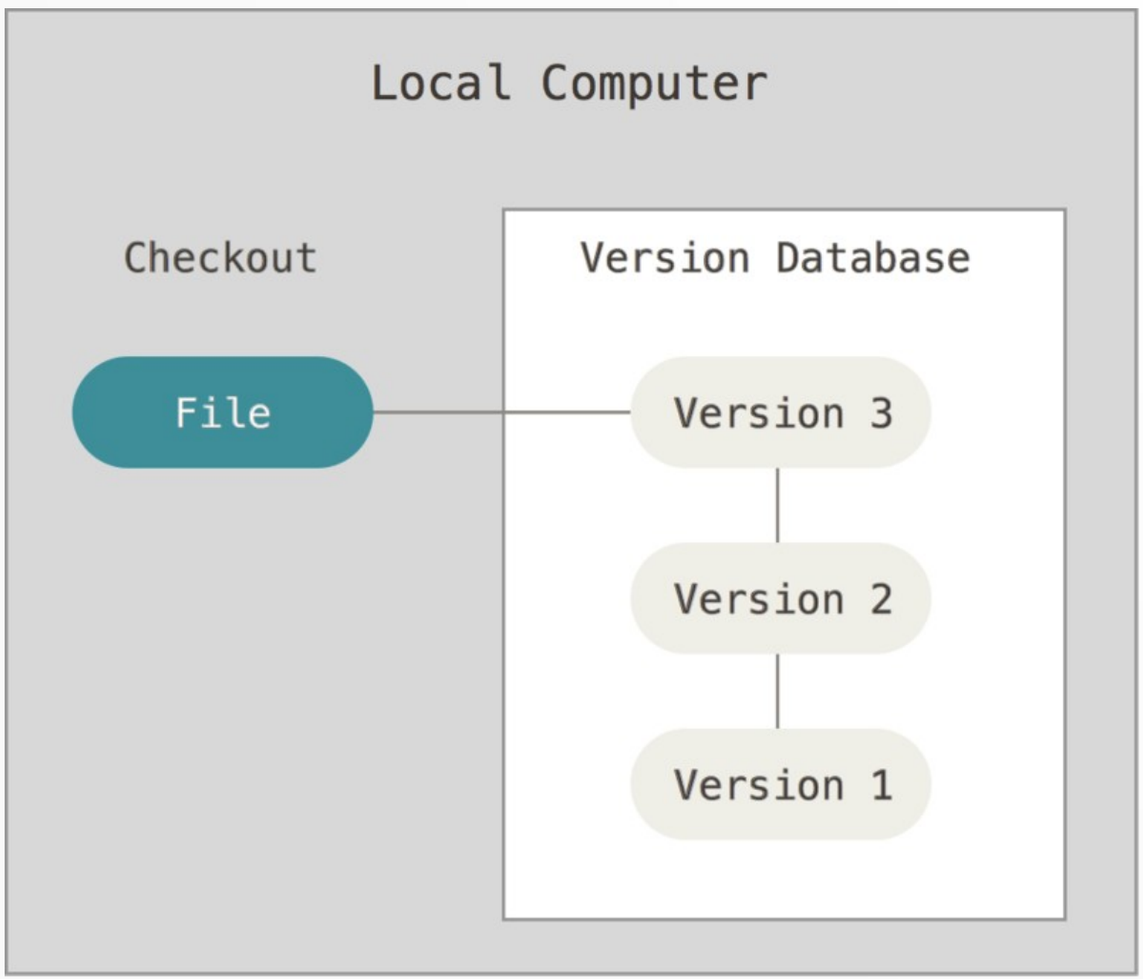
Content	what has changed?
Date	when did it change?
Author	who changed it?
Reason	why has it changed?

VCS does this

you enter this

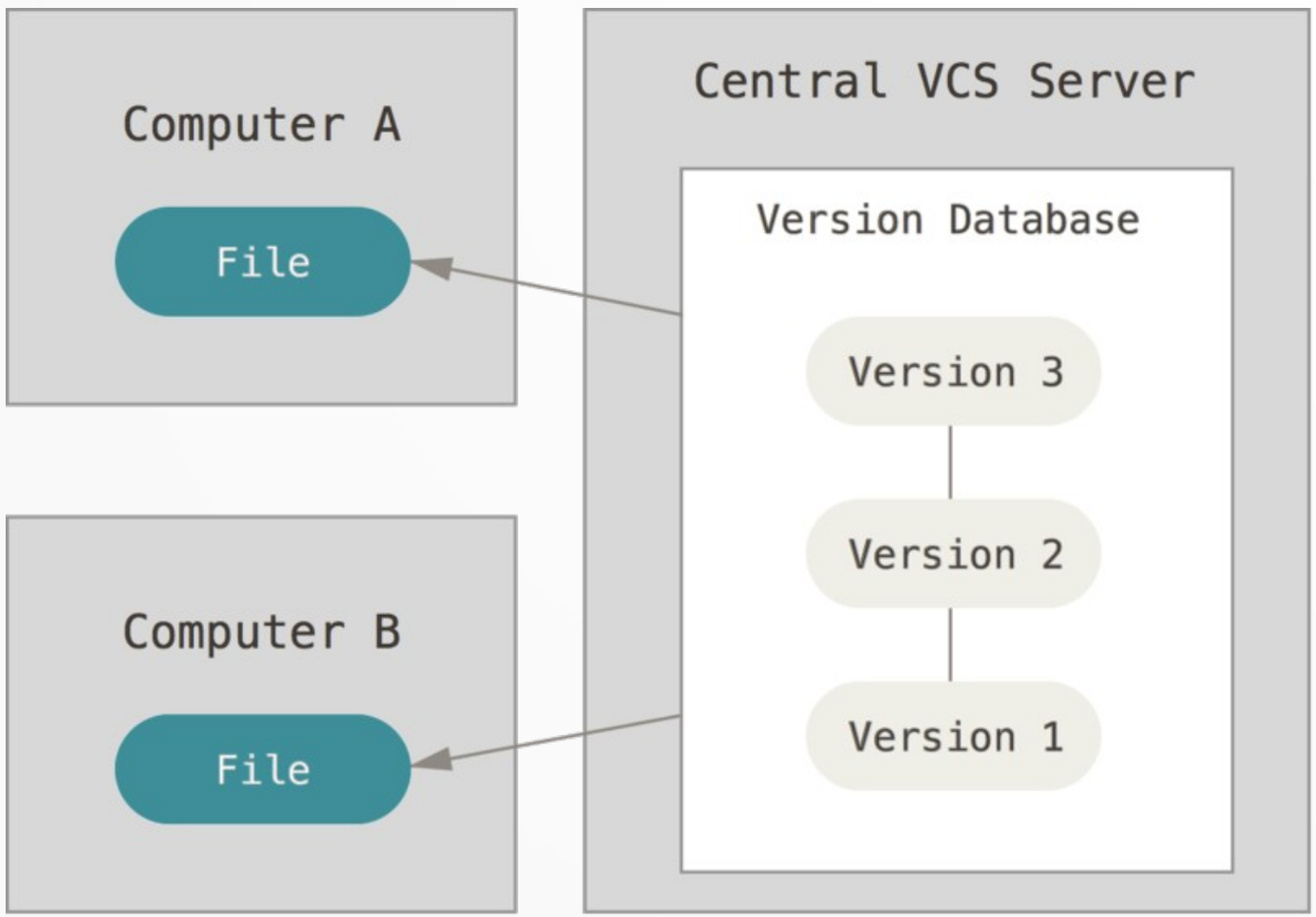
- The Reason (Commit Comment) should be meaningful and explanatory !!
- Don’t say how you have changed but why and what
  - Bad: bug-fix
  - Good: Correctly initialize variable x because of division by zero
- Separate subject from body with a blank line
- Limit the subject line to 50 characters
- Wrap the body at 72 characters

Local Version Control System



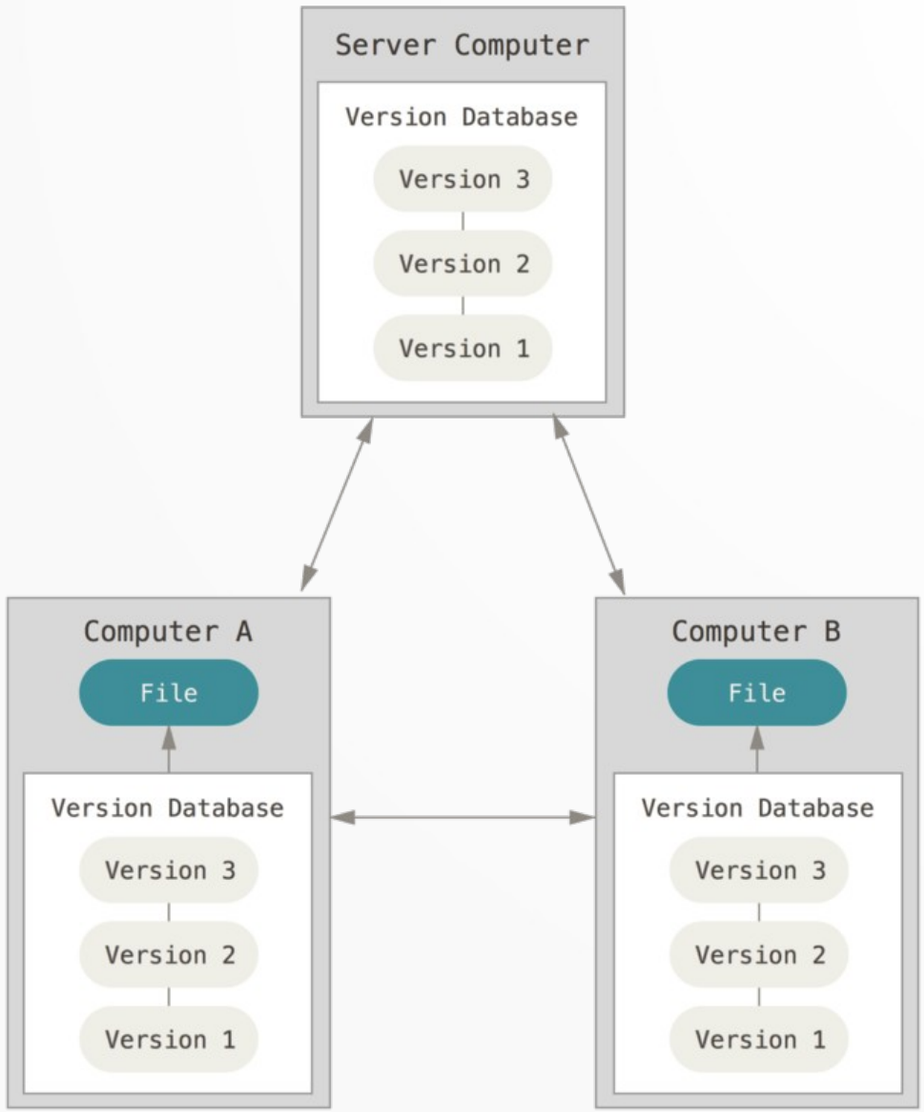
- Store differences between versions (deltas) for each file in a special format
- Less error prone than manually managing file versions
- E.g. RCS

Central Version Control System



- Allows a collaboration of different developers
- Fine grained access control is possible
- Central Server is a single point of failure
- Commits went always to the central server
- If the server is locally on your computer it works as local VCS
- E.g. CVS or Subversion

Distributed Version Control System



- Each computer has a copy of the full history
- Commits are done first locally which allows to save the full history of changes locally
- No single point of failure since every client copy is a full backup
- Depending on workflow synchronization can be complicated
- Most workflows use a central server
- E.g. Git, Mercurial

GIT

- Basic commands
- GIT Internal
- Branching
- Merging
  - Merge commits vs. Rebase

GIT

- Do you know how many GIT subcommands exist?
  - e.g. git add

GIT commands

- There are 145 git commands
  - Information from a talk by Scott Chacon
    - Co-founder of GitHub
    - https://www.youtube.com/watch?v=aoII_Rz0ZqY

GIT commands

- There are 145 git commands
  - Information from a talk by Scott Chacon
    - Co-founder of GitHub
    - https://www.youtube.com/watch?v=aoII_Rz0ZqY
- 82 Basic commands
  - 44 main commands (add, commit, push, pull, …)
  - 11 manipulators (config, reflog, replace, …)
  - 17 interrogators (blame, fchk, rerere, …)
  - 10 interactors (send-email, p4, svn, …)
- 63 plumbing (low level) commands
  - Check the presentation

GIT commands

- GIT add new commands but doesn’t remove the old ones
  - Available commands depend on the used GIT version
  - If you are working with various git versions you may know both
- Often different commands do the same thing
  - Revert changes to a file
    - **git checkout -- <file>** and **git restore <file>**
  - Change the working branch
    - **git checkout <branchname>** and **git switch <branchname>**

Workflow

- Local and central version control systems define a more or less fixed workflow
- With a distributed version control system many different workflows become possible
- Git is a very powerful toolbox to implement many different workflows
  - Good: very powerful and flexible
  - Bad: very powerful and flexible
- Before starting with a collaborative project define the used workflow

Repository

- The “Central Repository” is the official source of the project on a central server
  - GitHub or GitLab are well known repository providers
    - Allow to implement workflows
  - Everything else isn’t an official version !!!
- A “Fork” is a private clone (copy) of the official repository on the central server at a given point in time
  - Done using the services provided by GitHub or GitLab
  - Allow to do changes without affecting the central repository
  - Not synchronized automatically with official repository
  - Needed to integrate changes into the “Central Repository”
- The “local repository” or “working copy” is a clone from the central server on your local computer

Getting help

- GIT offers are very powerful builtin help
  - Get an overview about most important commands
    - **git help**
  - Get a list off all commands
    - **git help -a**
    - Can be used to confirm the number of commands
  - Get detailed help about a subcommand
    - **git help <subcommand>**
    - e.g. **git help blame**
- GIT Book
- Webpage

Git Configuration

- Configuration can be
  - Per user in ~/.gitconfig
    - **git config --global**
  - Per repository in *path_to_repository/*.git/config
    - **git config --local**
  - Use git attributes for a per directory config
  - Also a system wide config is possible
- Check config with
  - **git config --list**

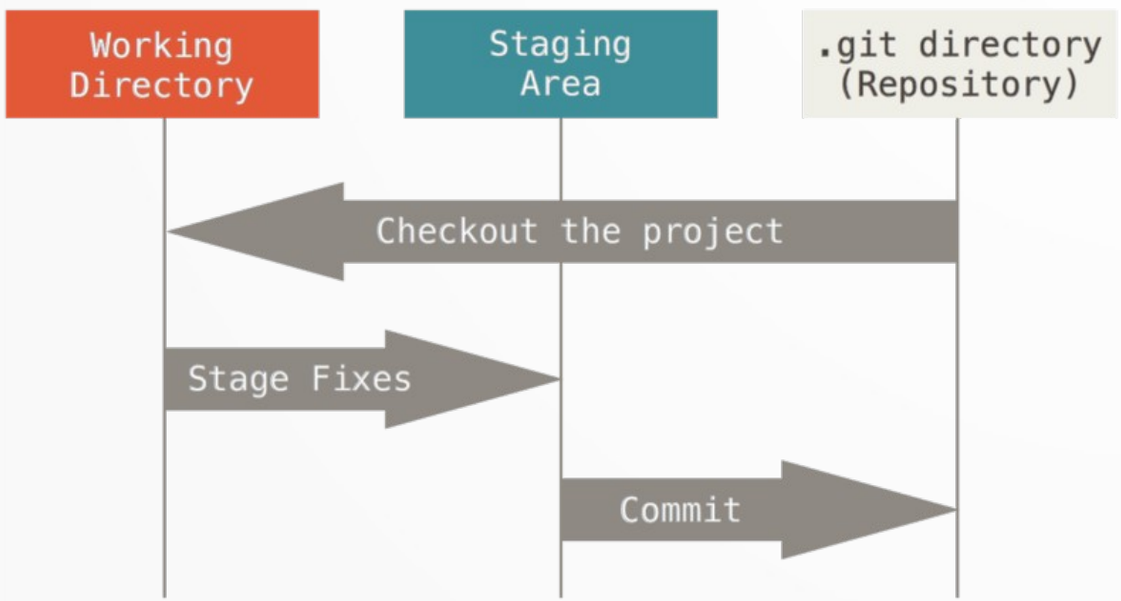
Advanced Git Configuration

- Define name and mail address which show up in the log message
  - **git config --global user.name “John Doe”**
  - **git config --global user.email “johndoe@example.com”**
- Define your preferred editor
  - **git config --global core.editor “joe”**
- Put output in columns
  - **git config –global column.ui auto**
- Sort your branches as function of time
  - **git config –global branch-sort -committerdate**
- Define aliases
  - **git config --global alias.st “status -bs”**
    - Status output with ***git st***
  - **git config –global alias.l “log --graph”**
    - History view with ***git l***

Git Clone

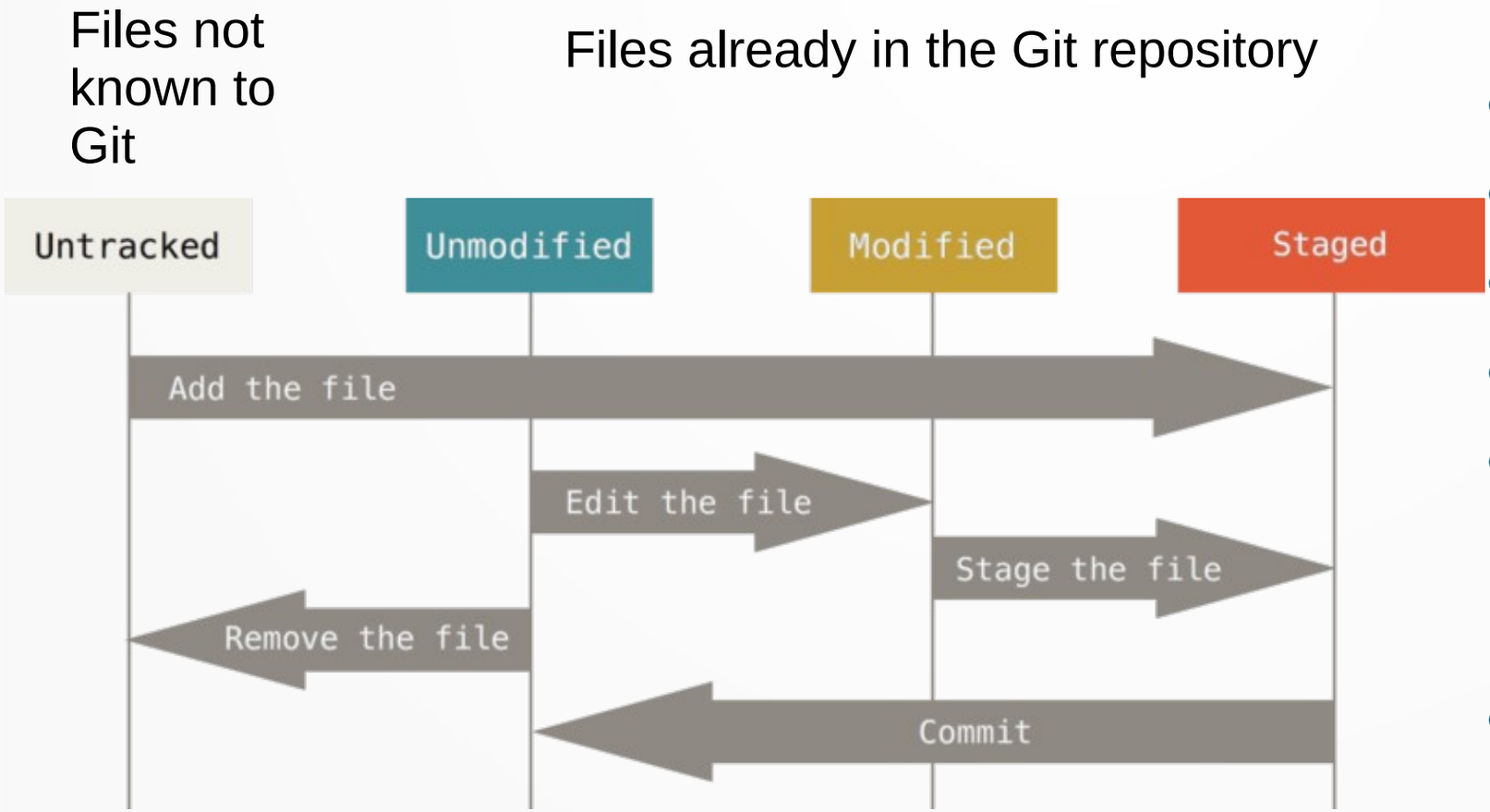
- Create a “working copy” of an existing git repository on your local computer
- Clone with different transfer protocols
  - HTTP
    - git clone https://gitlab.in2p3.fr/f.uhlig/base_project/
  - SSH
    - git clone git@gitlab.in2p3.fr:f.uhlig/base_project/
  - GIT
    - git clone git://gitlab.in2p3.fr/f.uhlig/base_project/
  - Filesystem
    - git clone /some_filesystem/base_project
  - If no branch is specified the default branch is available after cloning
    - Normally the branch is called “master” or “main”

Git Areas



- The working copy is a single checkout of one version of the project
- The staging area contains information what will go into your next commit
- The .git directory is the place where git stores the metadata and object database for your project.
- The .git directory is the local repository

Git Lifecycle



- Check current status
  - git status
- Add a file ***git add***
- Stage a file ***git add***
- Remove a file ***git rm***
- Rename a file ***git mv***
- Unstage
  - ***git restore --staged***
- Unmodify
  - ***git checkout –-***
  - ***git restore***

Diff

- View not yet staged changes
  - **git diff**
- View staged changes
  - **git diff –staged**
- View changes compared to a different branch
  - **git diff <branchname>**
  - e.g. **git diff upstream/main**
- View staged changes before each commit
  - To be sure what you are about to commit
  - You only commit files which are already staged

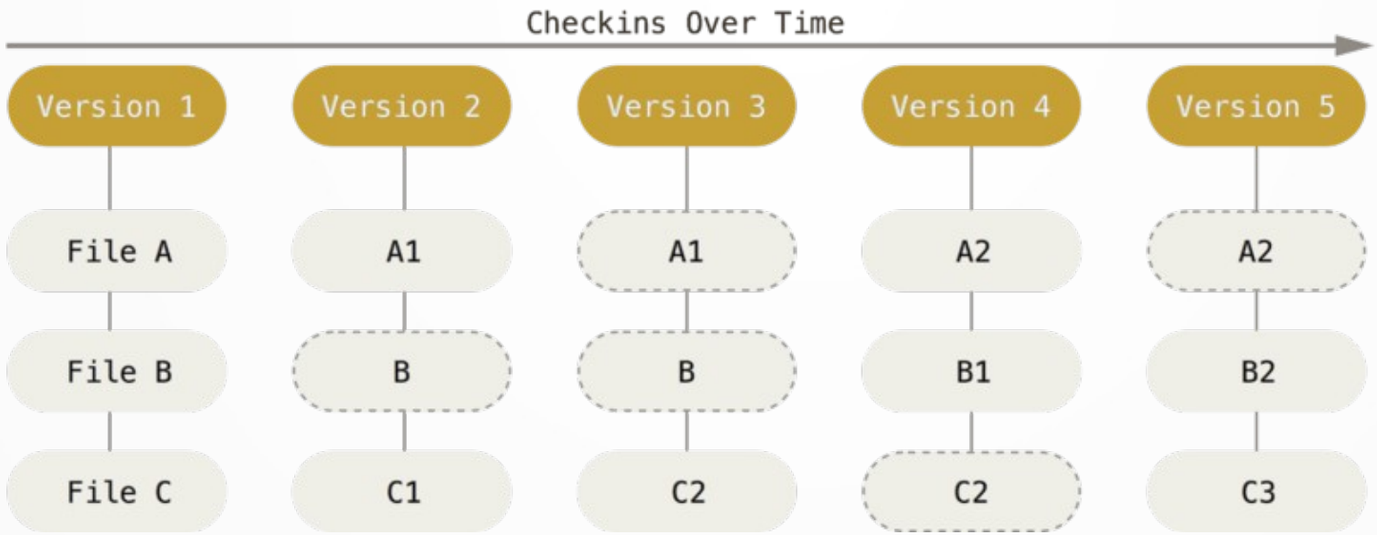
Commit

- Choose a good commit message
  - https://chris.beams.io/posts/git-commit
- git commit
  - Opens your editor to enter the commit message
    - Allows to type complex commit messages
    - Default editor if nothing else specified is “vi”
  - Will list all files which will be committed
- git commit -m ”Good commit message”
  - Shortcut for simple one-line commit message
- Up to now we are still on out local computer
  - git commit add changes only to the local repository

Commit Hash

- For each commit git calculates a unique hash value which identifies this commit
  - Hash value depends on the metadata and the patch set of the commit
  - Hash value depends also on the history of the current commit
  - It is not possible to change the history without leaving traces
  - If the history will be changed also the commit hashes will change
    - Avoid doing this on the “official repository”!!!
    - Avoid doing this with repositories you have shared with others!!
    - In your own workspace you can do whatever you like

Internal Representation



- Each commit is a snapshot of the git directory at the time of the commit
  - Git basically takes a picture of how your files look like at the time of the commit and stores a reference to this snapshot
- Don’t store unchanged files again but links to previous version

Other Useful Commands

- View the history
  - **git log**
- Get detailed information about a single commit
  - **git show *commit hash***
    - Changes + Metadata

Collaborative work

- So far everything work was only done with the local repository
  - Except the initial cloning of a repository
- How to work with collaborators?
  - How to make your changes known to others?
- GIT allows many different workflows for this purpose
  - Generate patches and send them by mail
  - Central server for synchronization
    - Most common ones are GitHub and GitLab

Git Commit vs. SVN Commit

- A “svn commit” send the local changes to the central server
  - Creates a new revision
- A “git commit” documents your local change history
  - Can be very granular
  - Documents what you did during your development which includes also all mistakes
- The local commit history can be changed
  - Squash commits
  - Reorder commits
  - Change commit messages, time, author, …
- In public branches the commit history must not be changed

Remote repositories

- Remotes are names which refer to other git repositories and are valid only for the working copy
- Show remote repositories
  - **git remote -v**
- *origin* is the default remote repository after cloning
- Add new remote repository
  - **git remote add <name> <url>**
- Remove a remote repository
  - **git remote remove <name>**

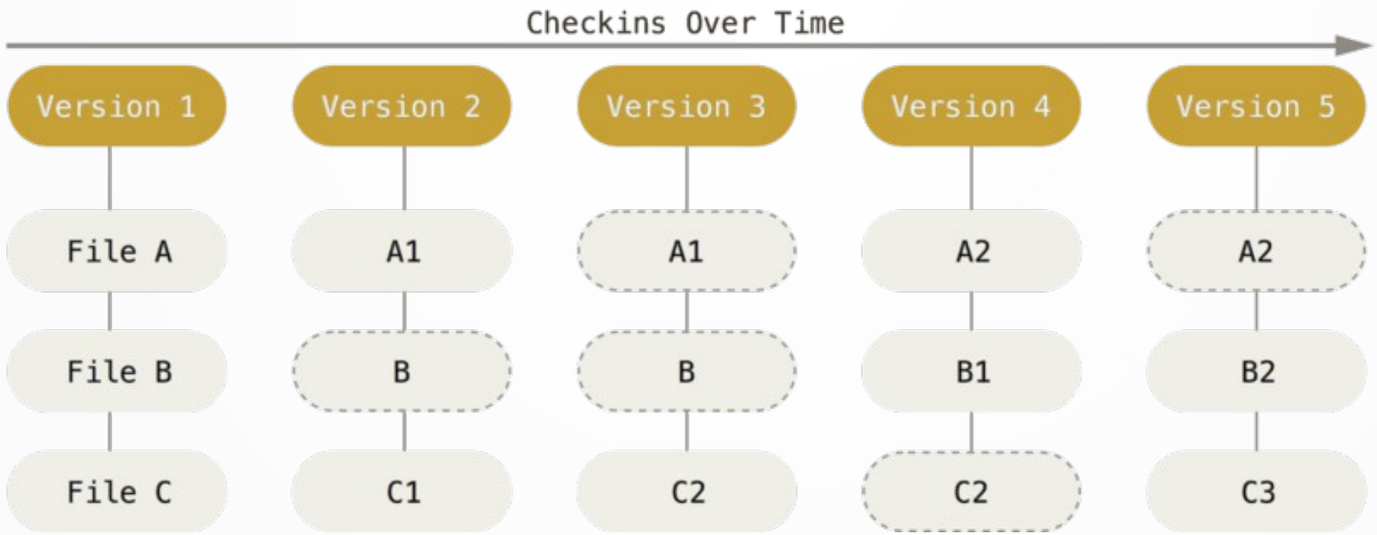
Interacting with Remotes

- Download all objects (e.g. commits) and refs (e.g. tags) from the remote repository which are not in your local repository.
  - **git fetch <remote name>**
  - This command does not merge the changes with your local work directory !!
- Upload changes from your local repository to the remote repository
  - **git push <remote name> <branch name>**
  - **git push <remote name <tag name>**
  - **git push <remote name> <local branch name>:<remote branch name>**

Tagging

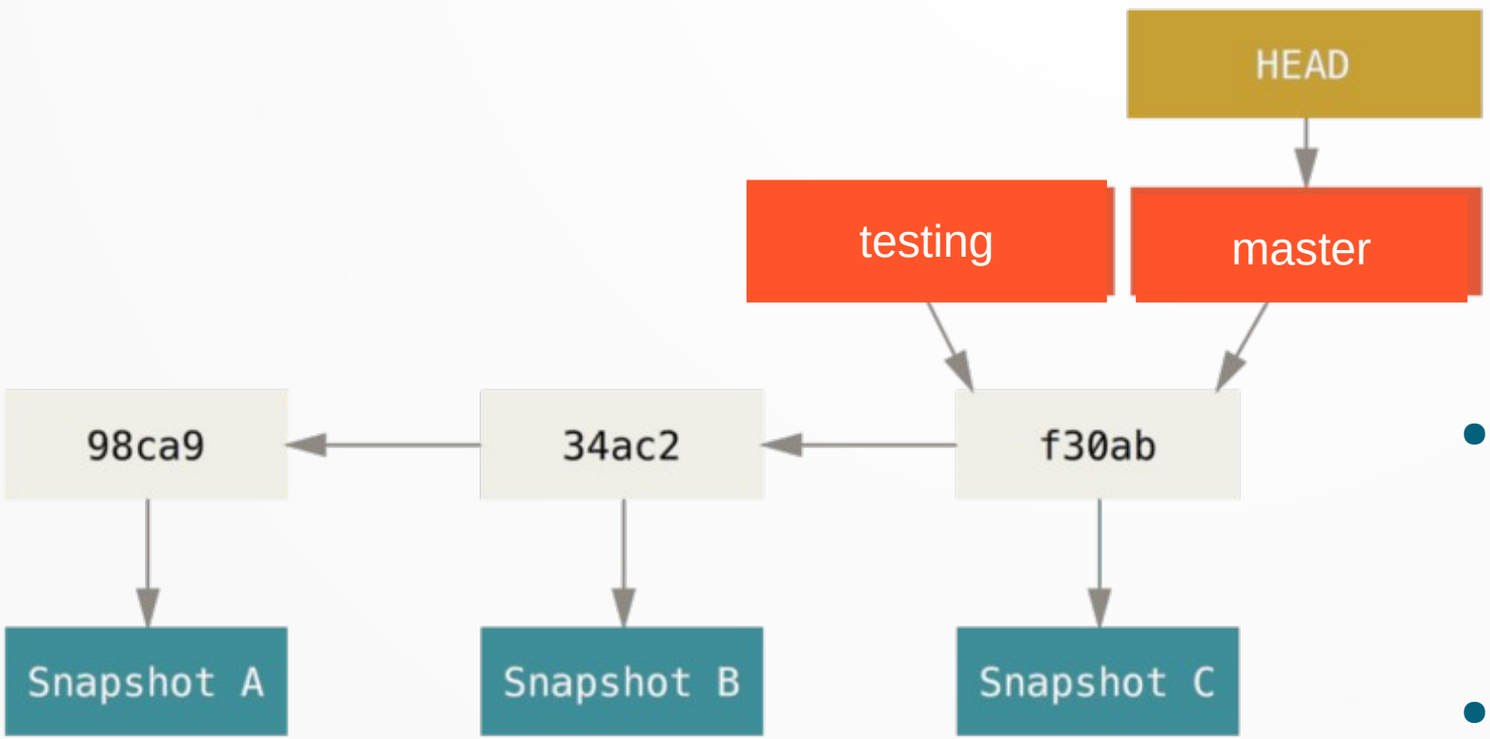
- A tag is an arbitrary repository-local name that points to a commit hash
  - Better to remember than a commit hash ;-}}
- A tag is used to define important points in the project history
  - Usually projects apply the versioning scheme via tags
- List tags
  - **git tag -l**
- Create an annotated tag
  - **git tag -a name -m”Major release”**
- Delete tag
  - **git tag -d name**
- Push tag
  - **git push <remote> <tag name>**
  - **git push <remote> --tags**

Internal Representation



- Each commit is a snapshot of the git directory at the time of the commit
  - Git basically takes a picture of how your files look like at the time of the commit and stores a reference to this snapshot
- Don’t store unchanged files again but links to previous version

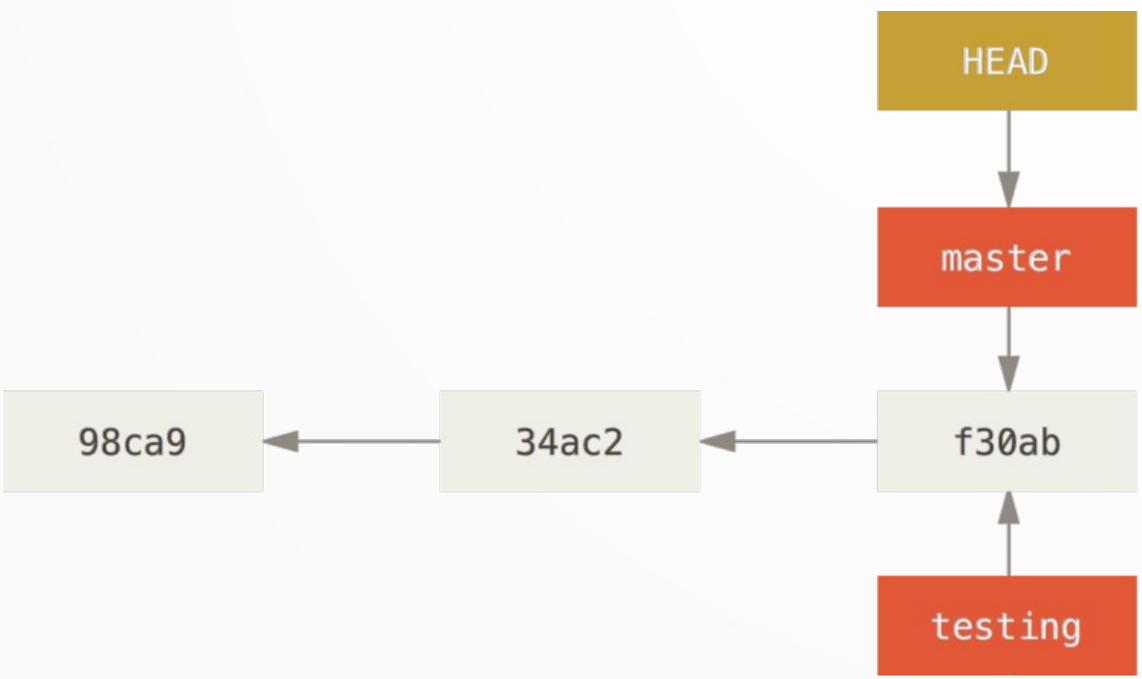
Branches



- Create a new branch from the current hash value
  - **git branch <name>**
  - e.g.

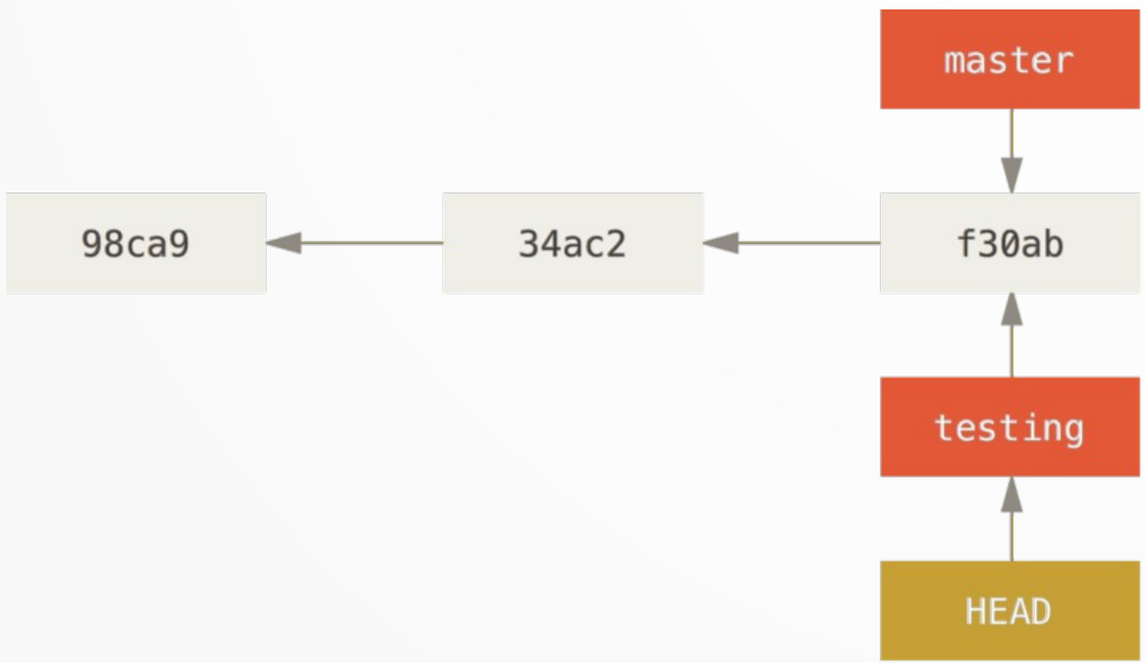
    git branch testing
- Creates a new pointer to the same commit you are currently on
- HEAD is a tag which points to the branch you are currently on

Branches



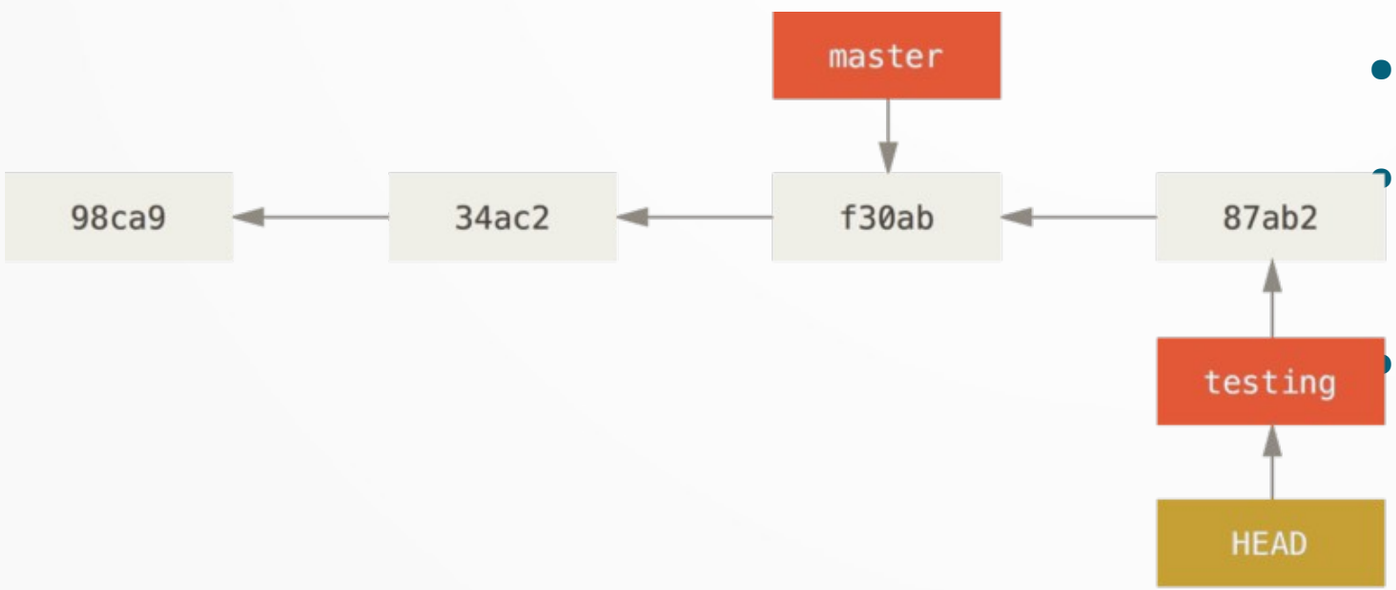
- Now switch to the newly created branch
  - **git checkout testing**
- Moves HEAD to point to the testing branch

Branches



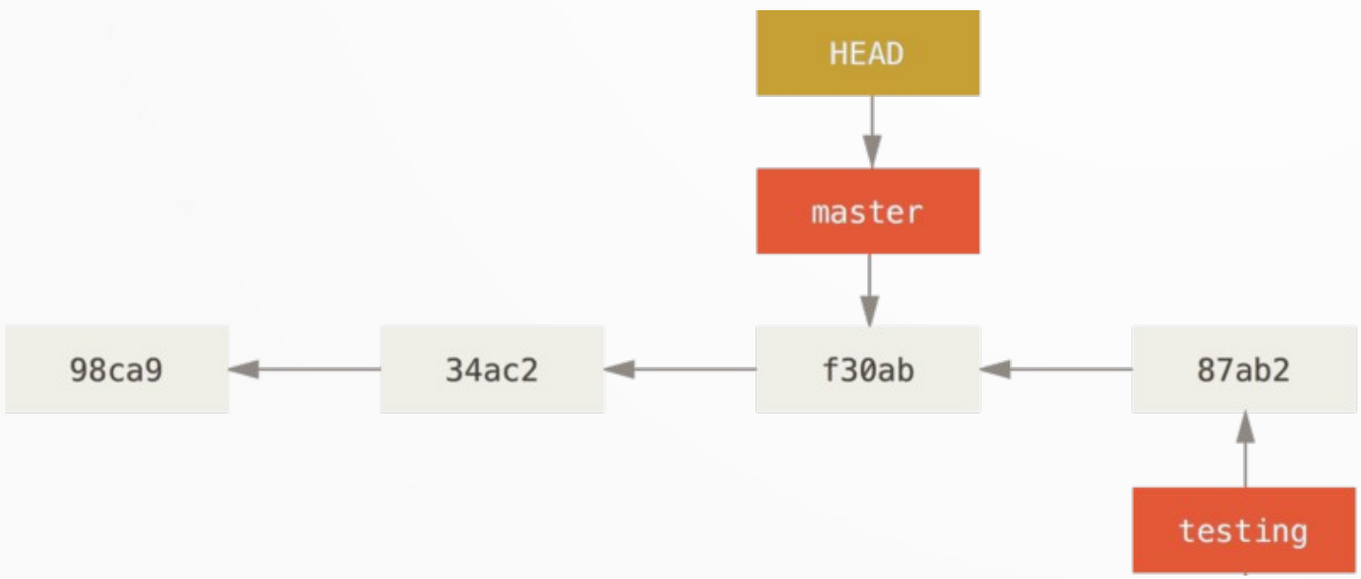
- Moves HEAD to point to the testing branch
- Let's change a file and commit
  - **git commit -a -m "change"**

Branches



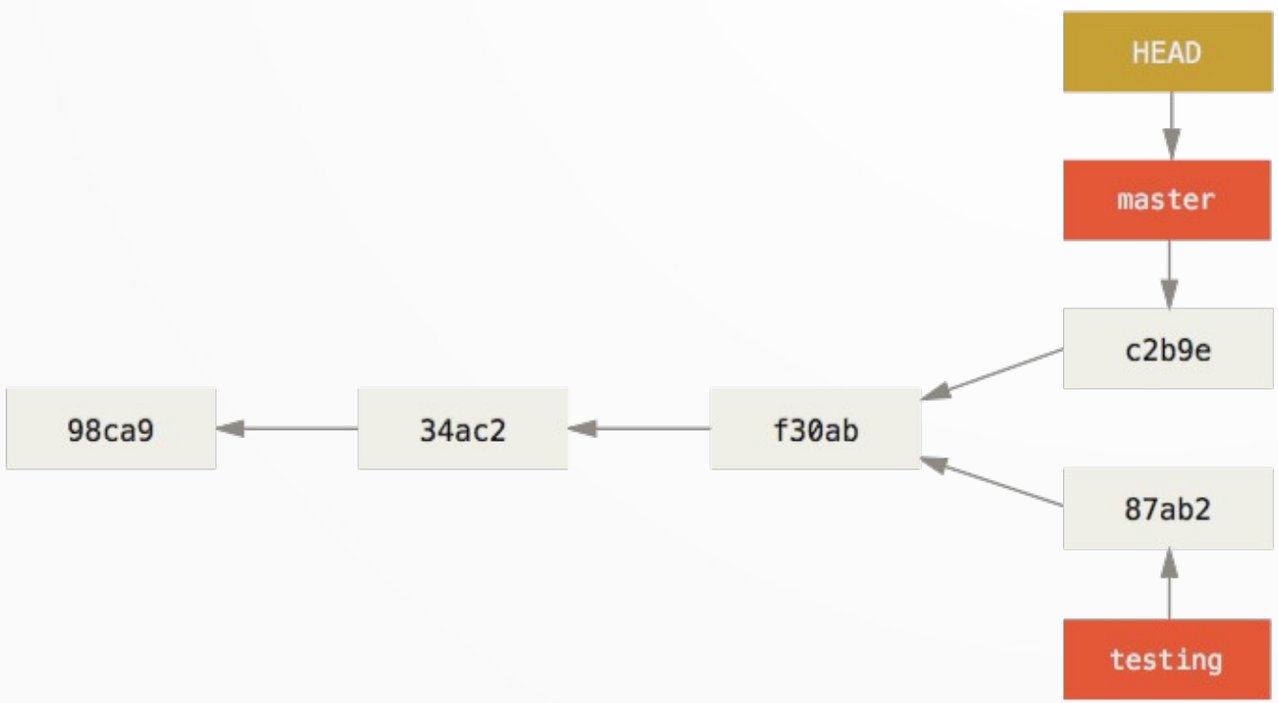
- Let’s change a file and commit
  - **git commit -a -m “change”**
- Testing has moved forward
- Master still points to old commit
- Let’s switch back to master
  - **git checkout master**

Branches



- Let’s switch back to master
  - **git checkout master**
- Now HEAD points to master
- Also reverted files in your working directory back to the snapshot master points to
- Let’s now commit something in master
  - **git commit -a -m “change 2”**

Branches

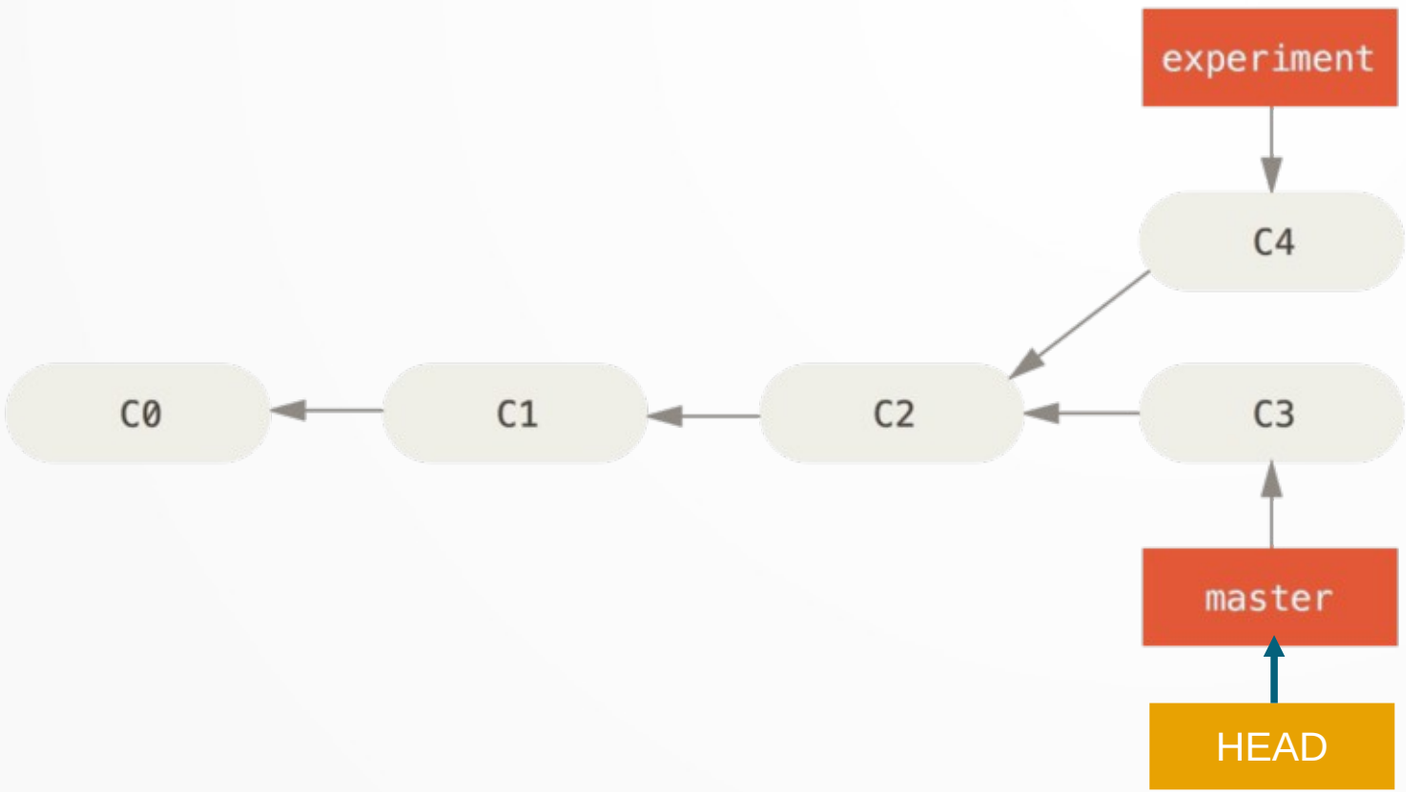


- Let’s now commit something in master
  - **git commit -a -m “change 2”**
- Now your history has diverged
- How to get the changes from testing into master?

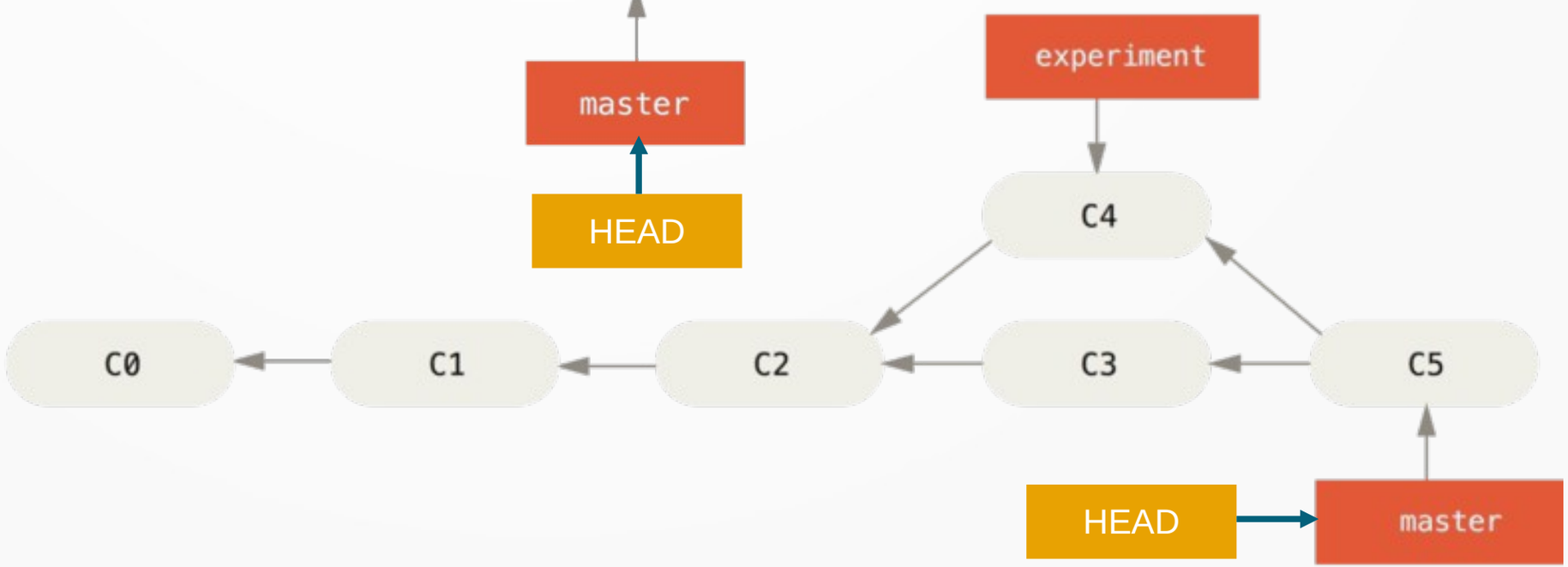
Merge vs. Rebase

- Two different ways to integrate the changes from one branch into another
- A ***git merge*** performs a three way merge between the two latests branch snapshots and the most recent ancestor of the two, creating a new commit
  - A merge has two parent commits
- With a ***git rebase*** you take all the changes done in one branch and reply them onto another branch
  - A rebase has only on parent
- Rebasing results in a clean linear history
  - Personally I prefer rebasing so I will introduce later a workflow using rebasing

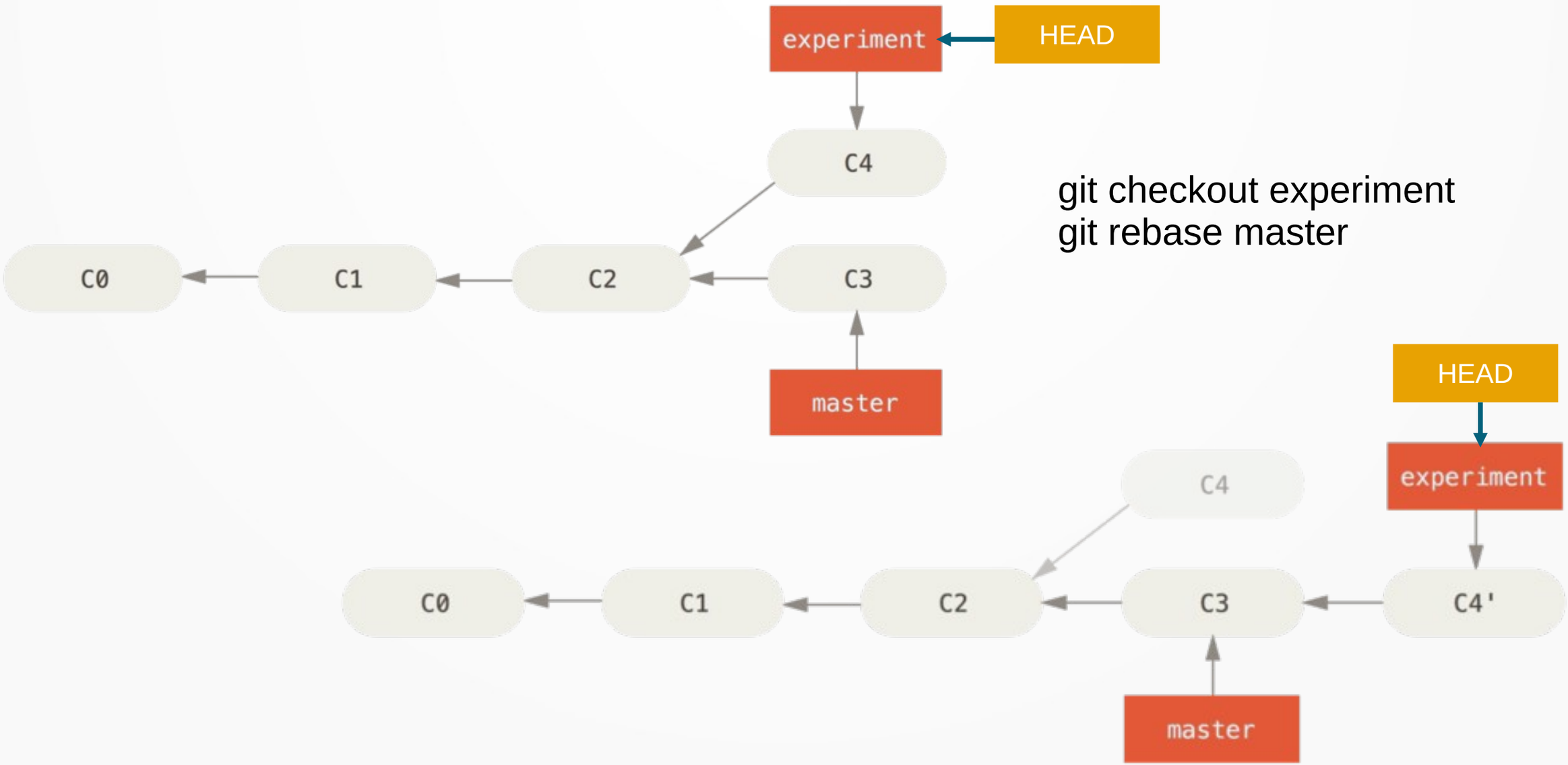
Merge



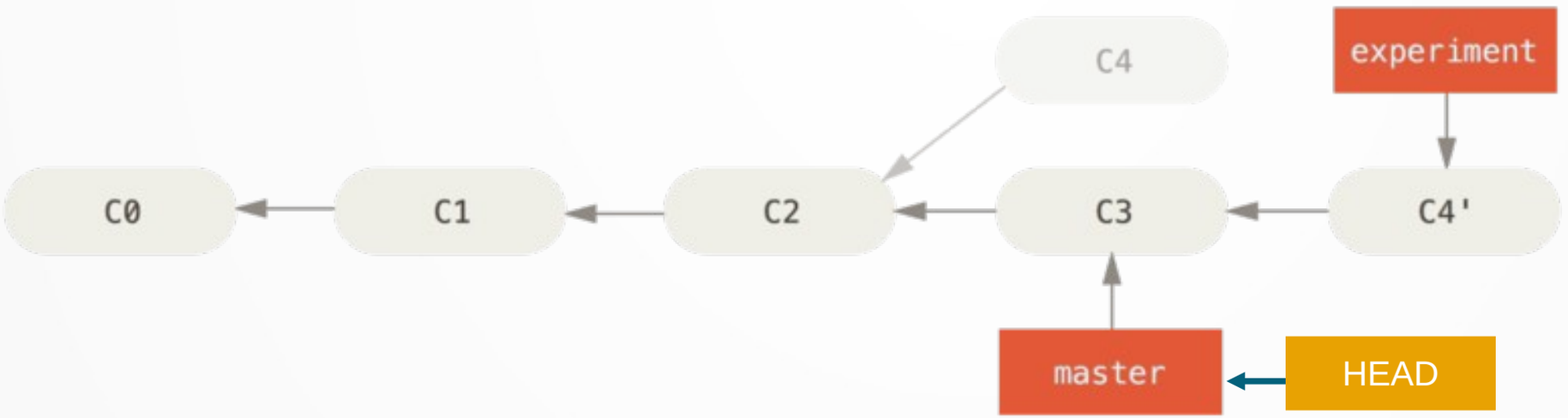
git checkout master
git merge experiment



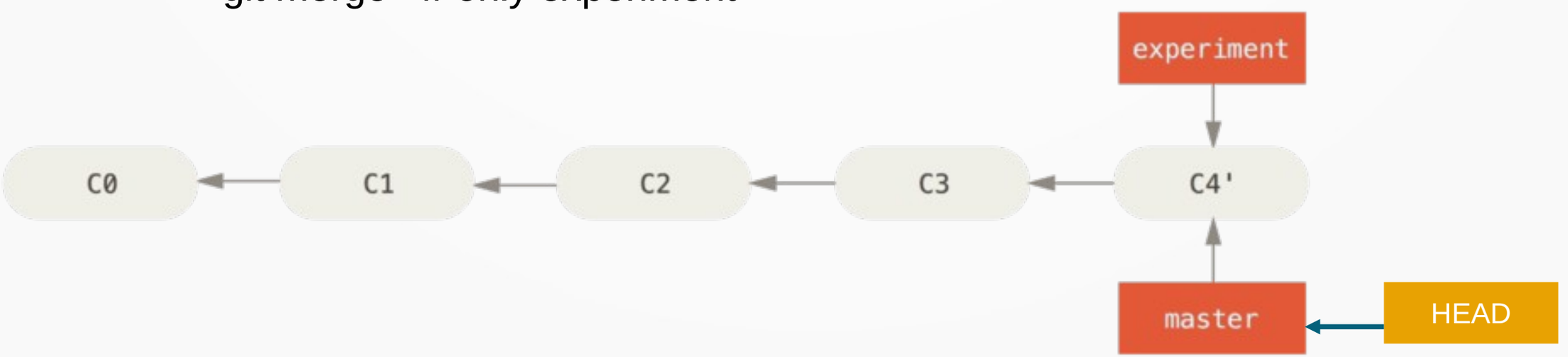
Rebase


experiment
HEAD
C4
C0
C1
C2
C3
master
git checkout experiment
git rebase master
HEAD
experiment
C4
C0
C1
C2
C3
C4'
master

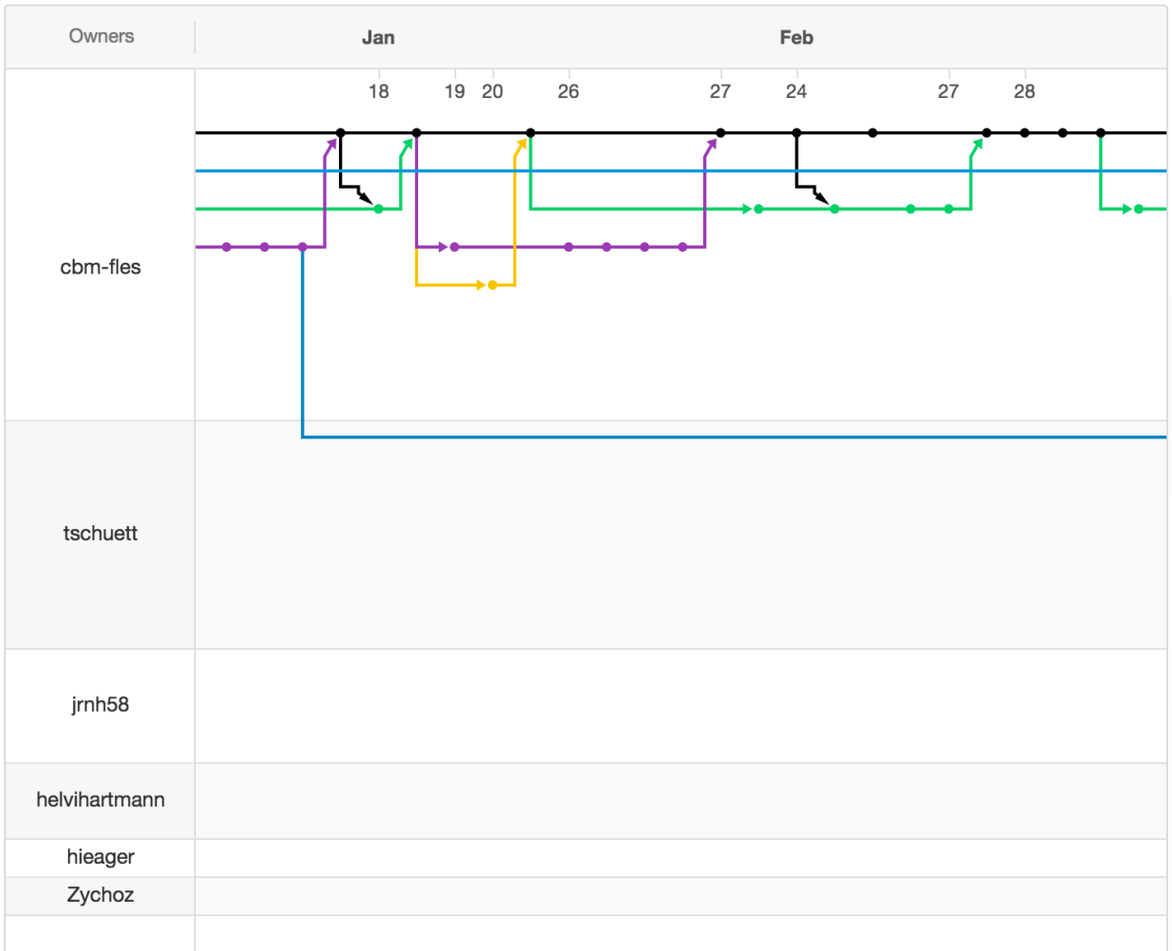
Rebase II

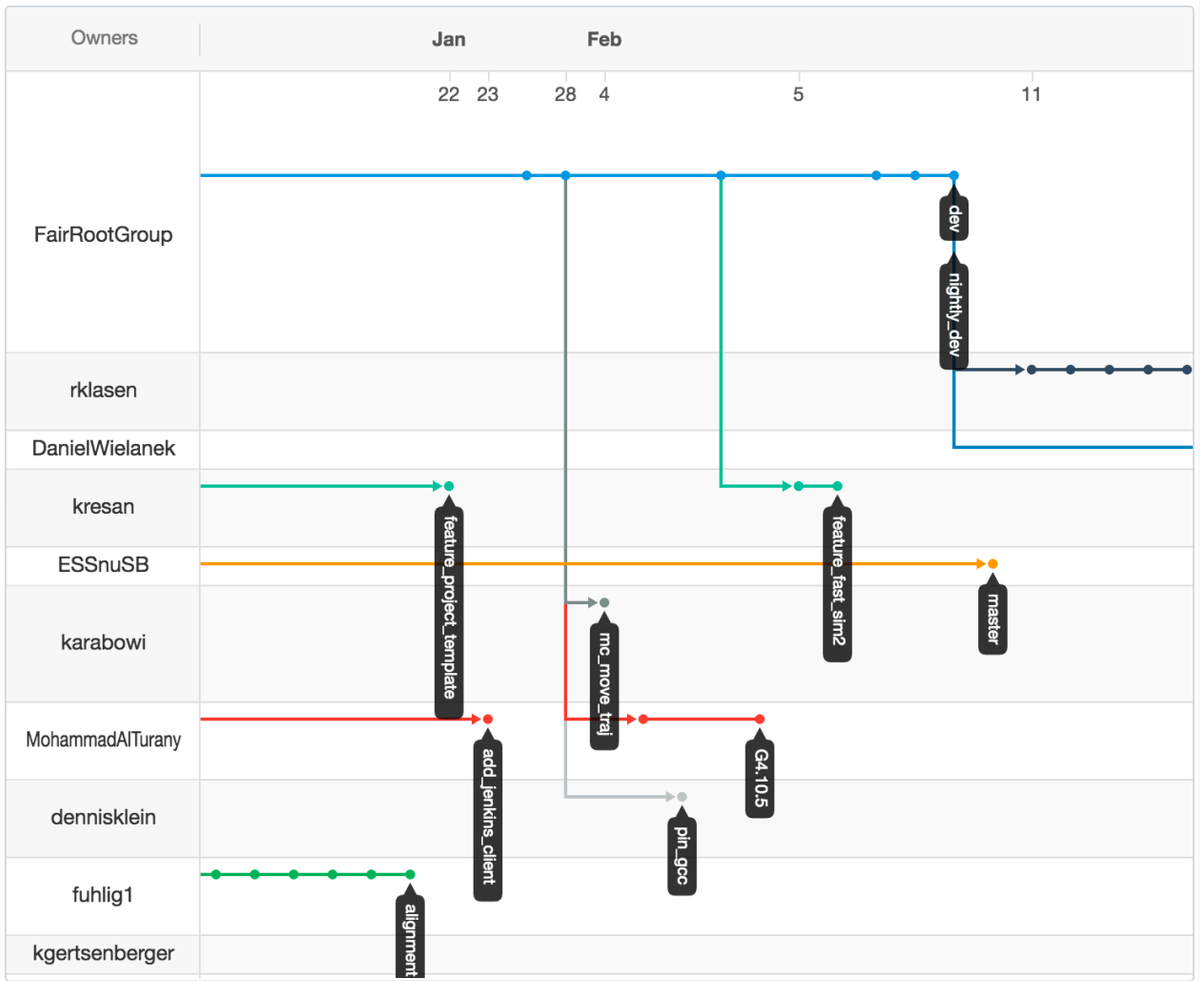


git checkout master
git merge –ff-only experiment



Repository graphs

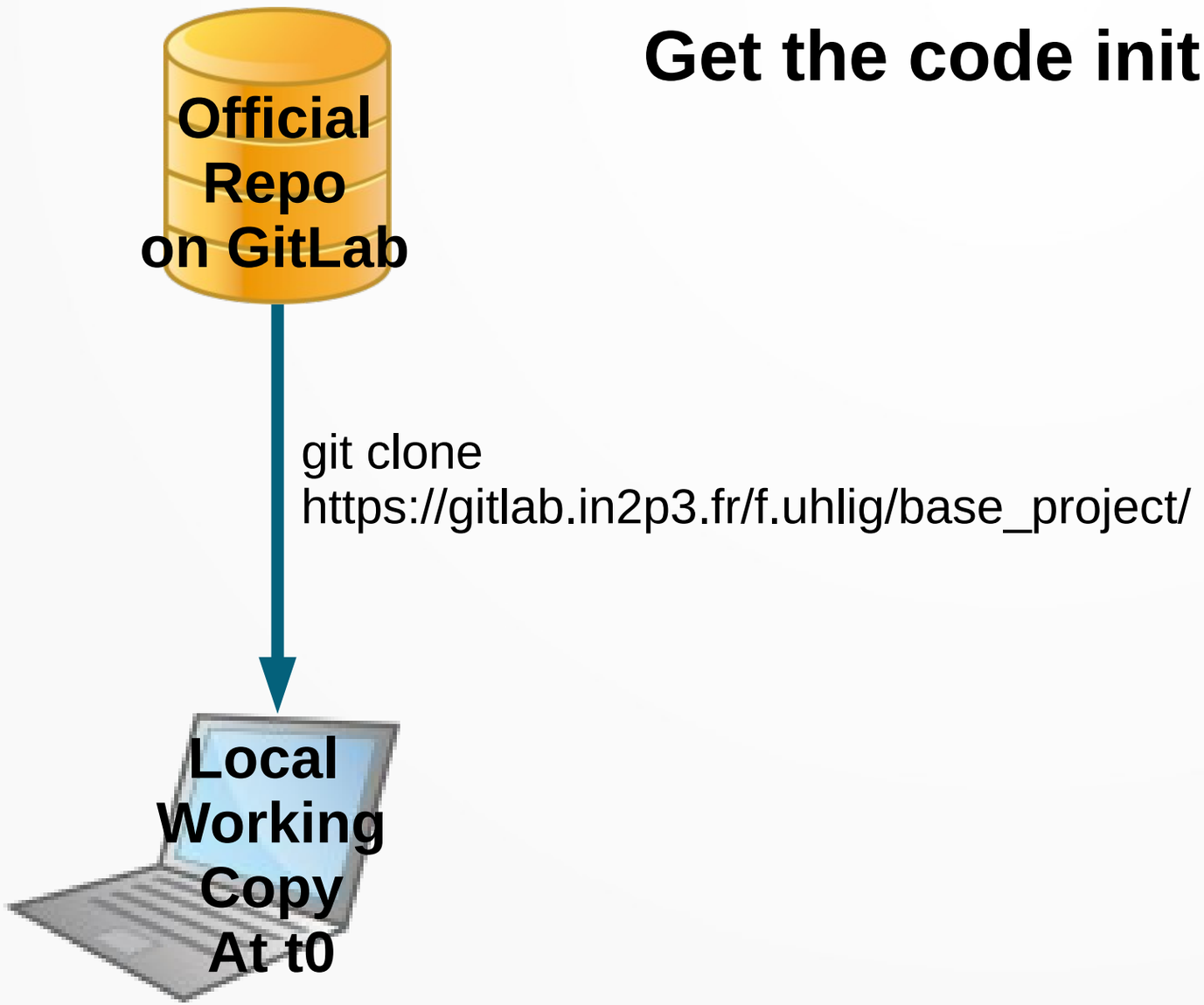




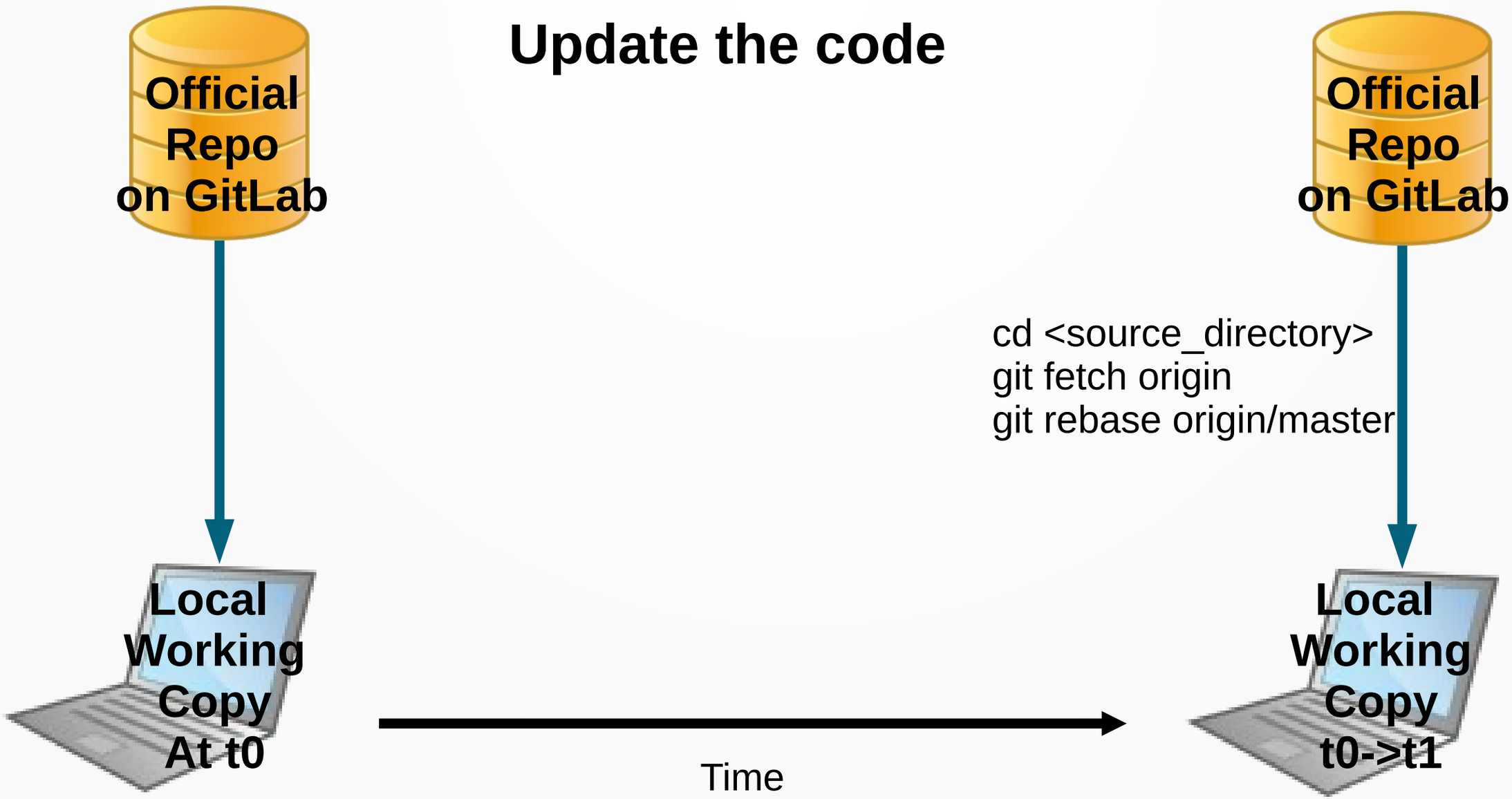
Naming Conventions

- User
  - Person who only intends to download and use the project source code
- Developer
  - Person who intends to contribute to the official project repository
- Manager
  - Person who manages the official project repository
  - I am not sure if this will be covered during this week

Workflow For Users



Workflow For Users



Summary For Users

- Either get the code initially or update the local working copy to the latest state
- No code changes are done

Workflow For Developers

- Developers get the code in the same way as the users
- Developers get updates in the same way as the users
- How do developers get their local changes back into the “Official Repository”?
  - Nobody has direct write access to the official repository!!!

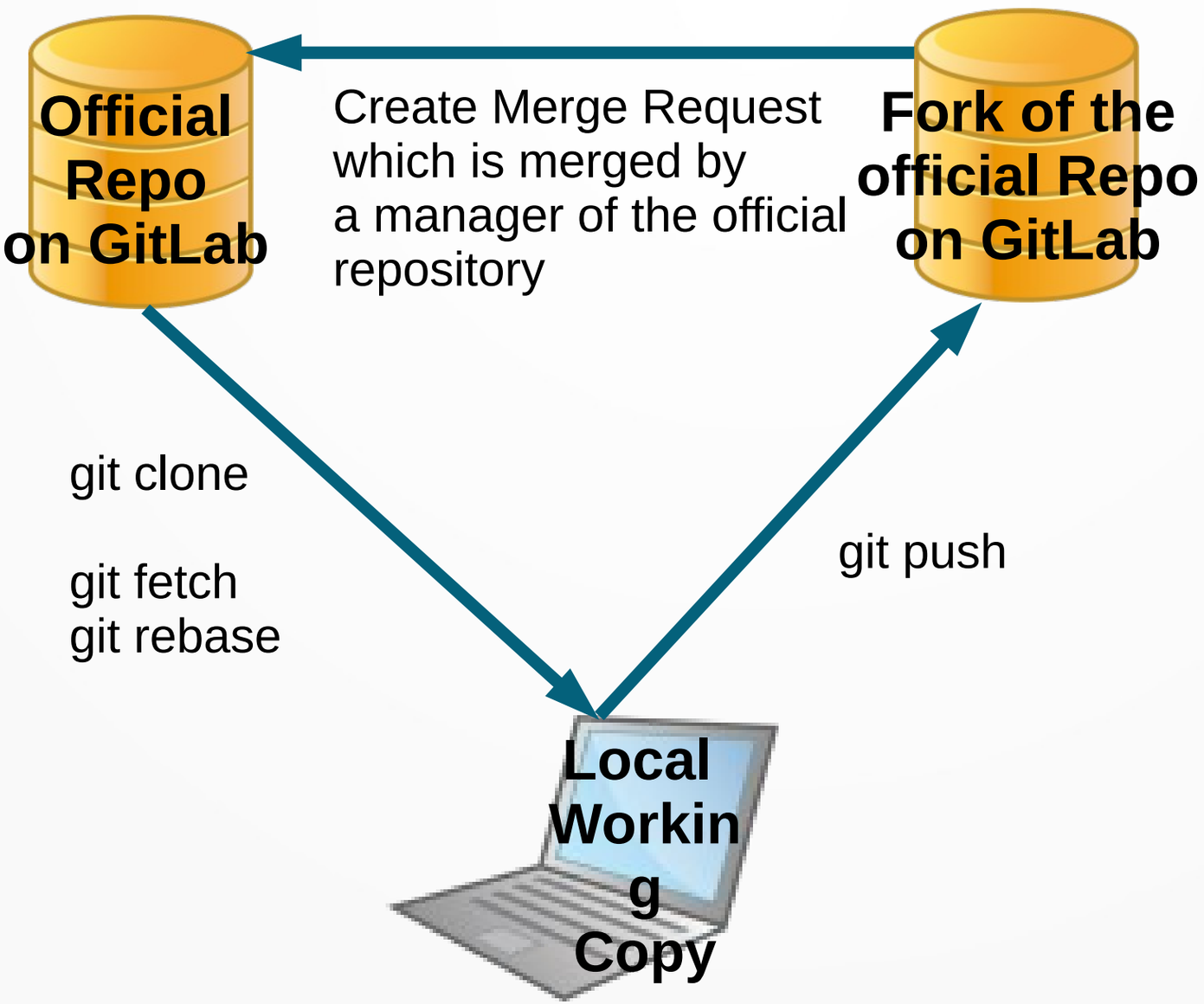
Workflow For Developers

Commit the code to the official repository

Workflow For Developers

- If the direct way is blocked we have to use a bypass
- The following slides show the proposed GIT workflow
  - Fork and Merge Workflow

Workflow For Developers Final Picture

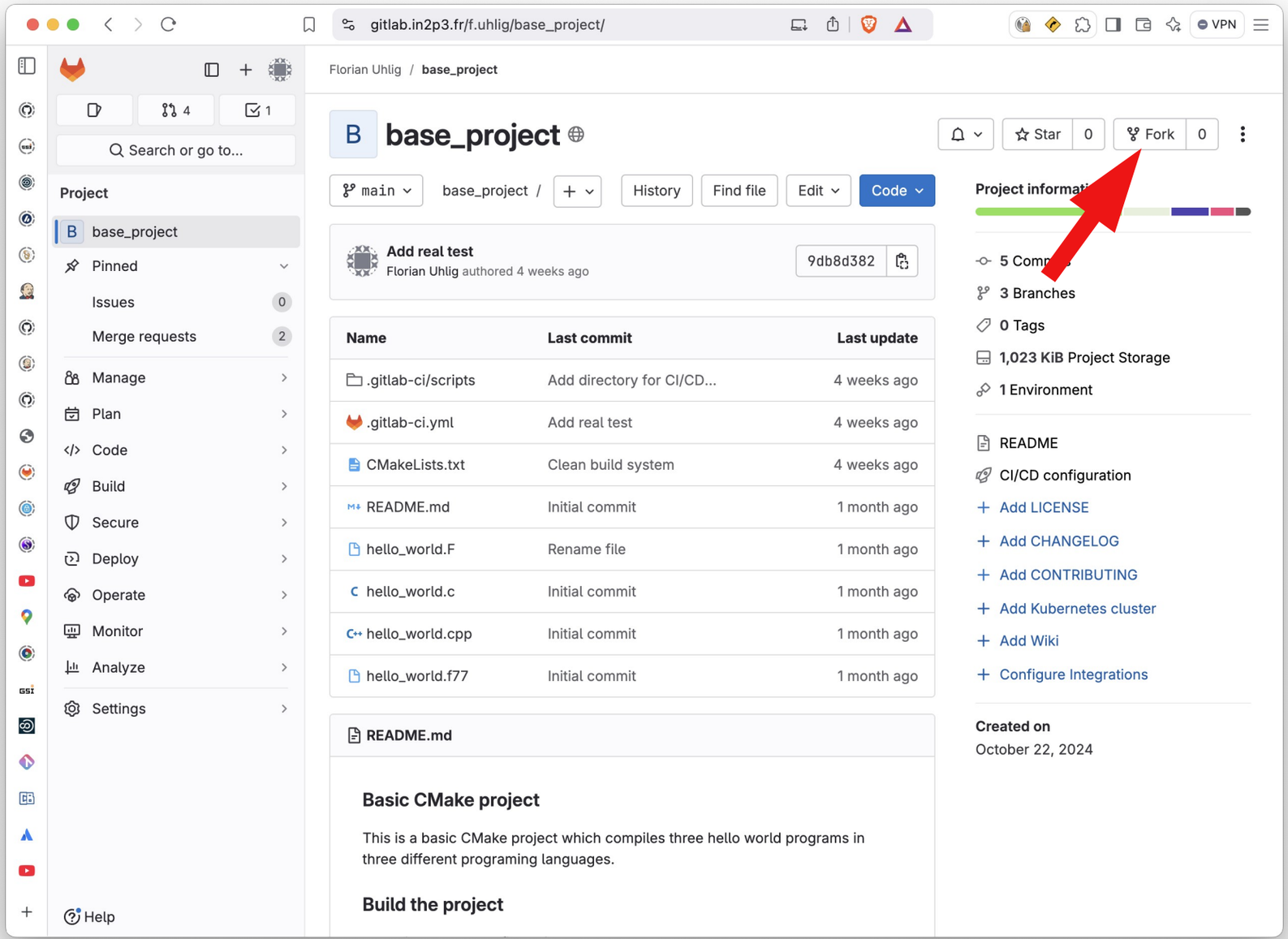


User → Developer

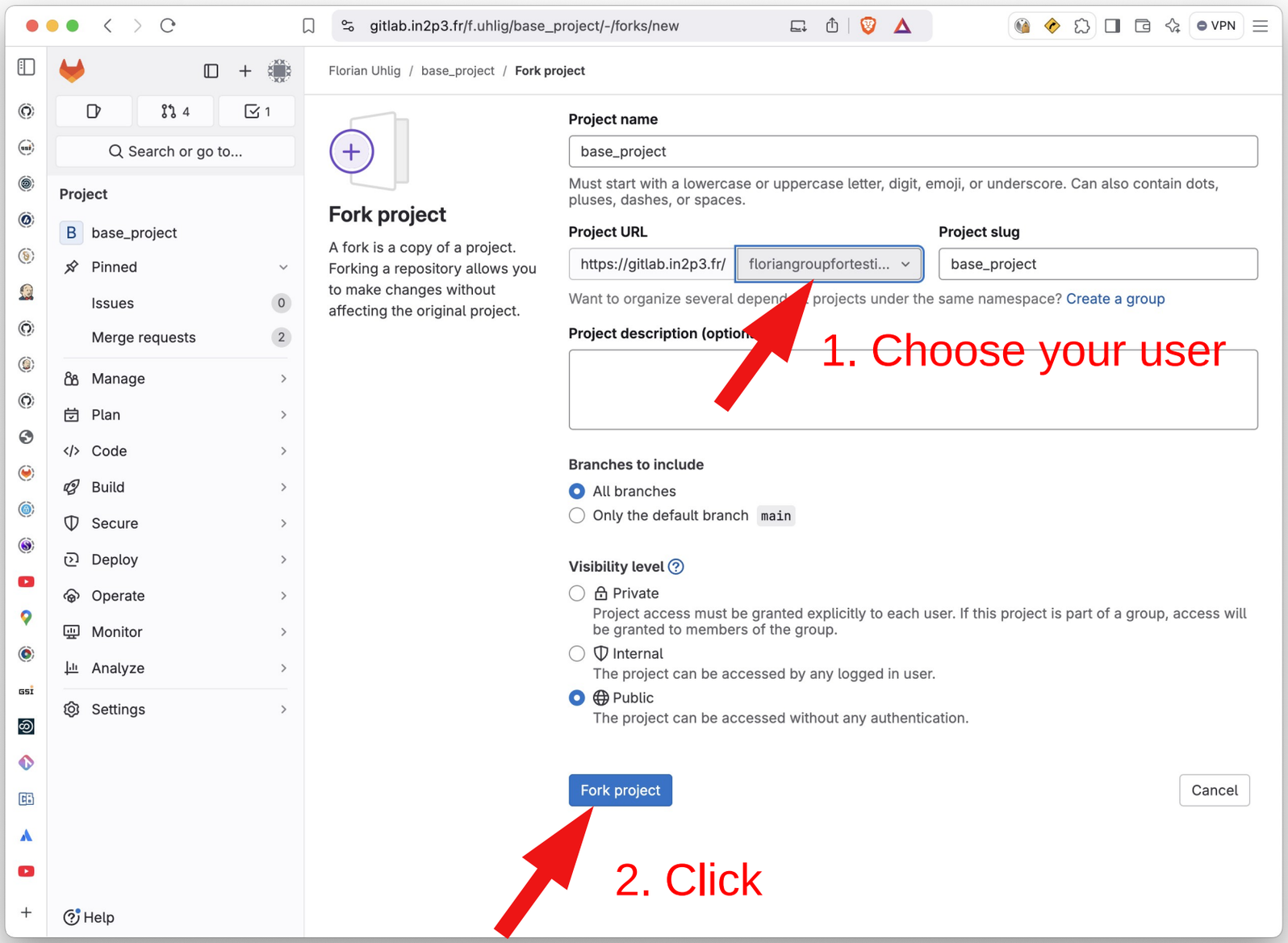
- How to get changes into the repository?
- Create a copy of the official repository on the GitLab Server in your user account
  - Creating a fork of the official repository
    - Forking is a server side copy of the project
  - Has to be done only once
  - Since this is done using the GUI of the web server some screenshots are attached
  - Navigate to the official repository of a small test project at

    https://gitlab.in2p3.fr/f.uhlig/base_project/
    - Click on the “fork” button
      - Creates a copy of the official repository in your user account

Fork The Repository



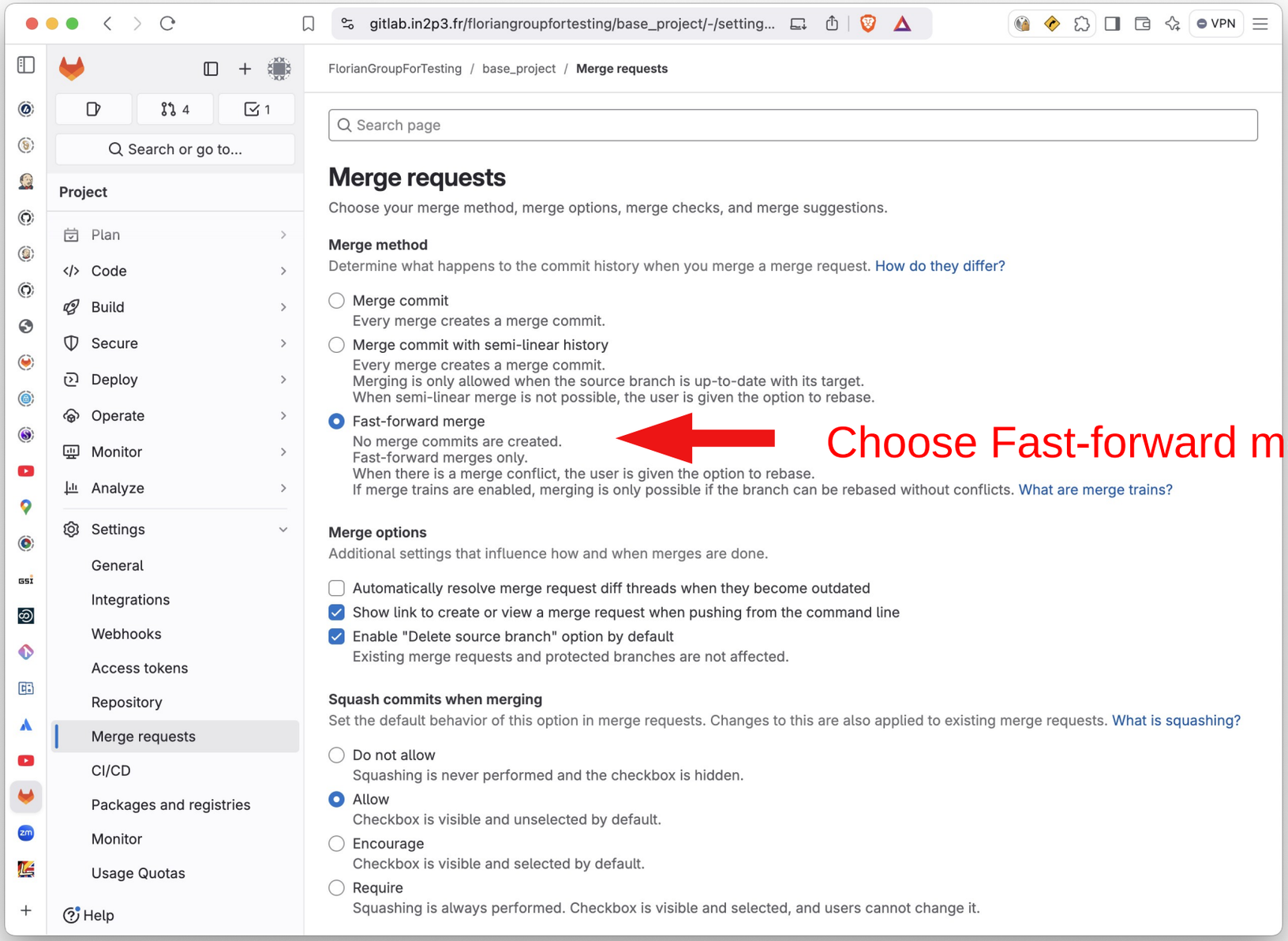
Fork The Repository II



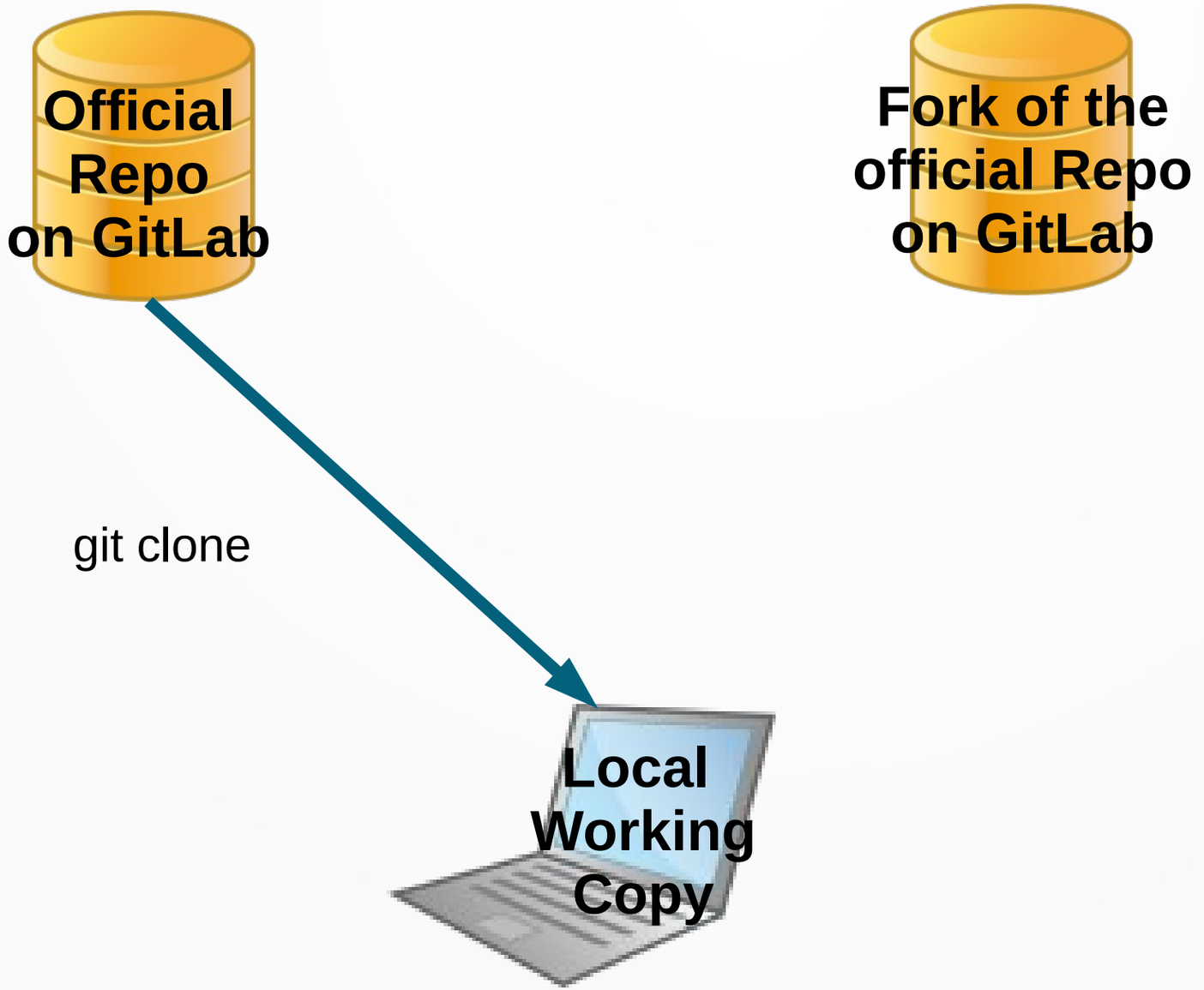
Project configuration

- Since we want to use the rebase workflow one needs to choose the correct merge method
  - Otherwise you may introduce merge commits in your fork which can’t be merged in the official repository
- All other settings are optional
  - Take your time to go through the list of options

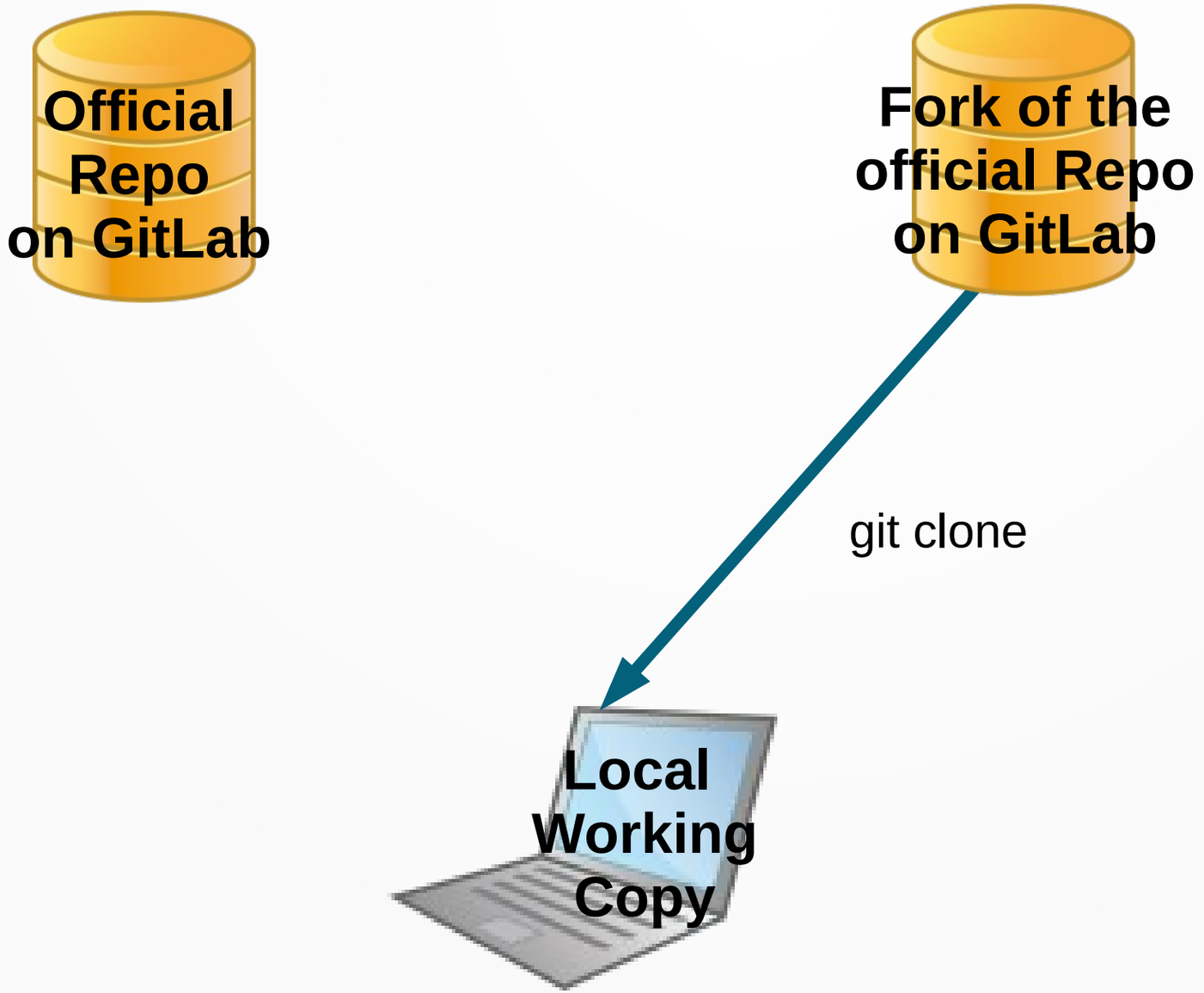
Choose the correct settings



After Creating the Fork



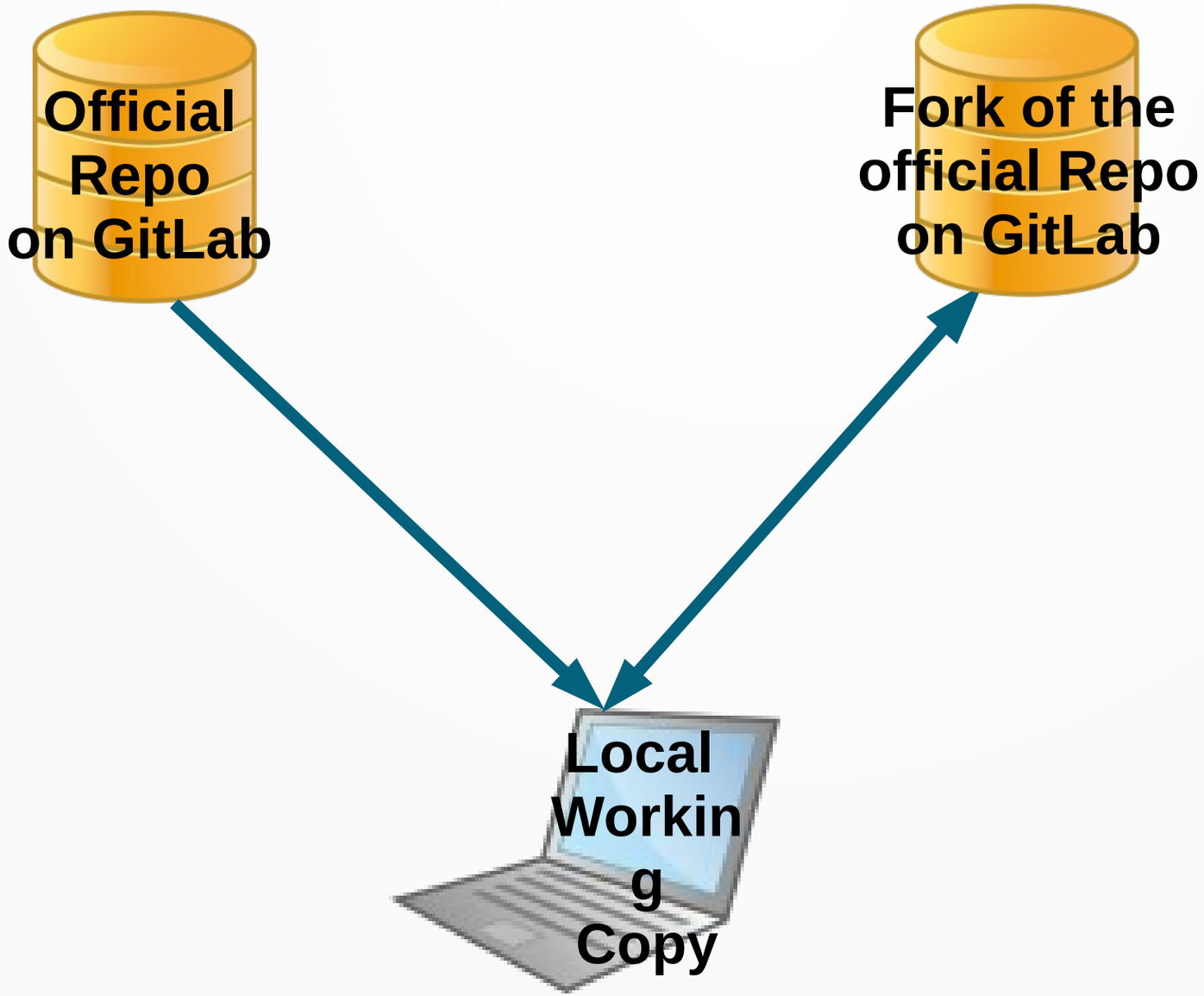
After Creating the Fork



Naming Conventions

- The repository which was used for the clone of the local working copy is named “origin”
  - Check where it points to
    - **git remote -v**
- “origin” is only a name
- “origin” may point either to your fork or to the official repository
  - Very unfortunate situation since people use the same name for two completely different things
- Everybody should use the same convention
  - Change the name
    - **git remote rename <old> <new>**
  - Use “upstream” for the official repository

Connecting the Second Repository

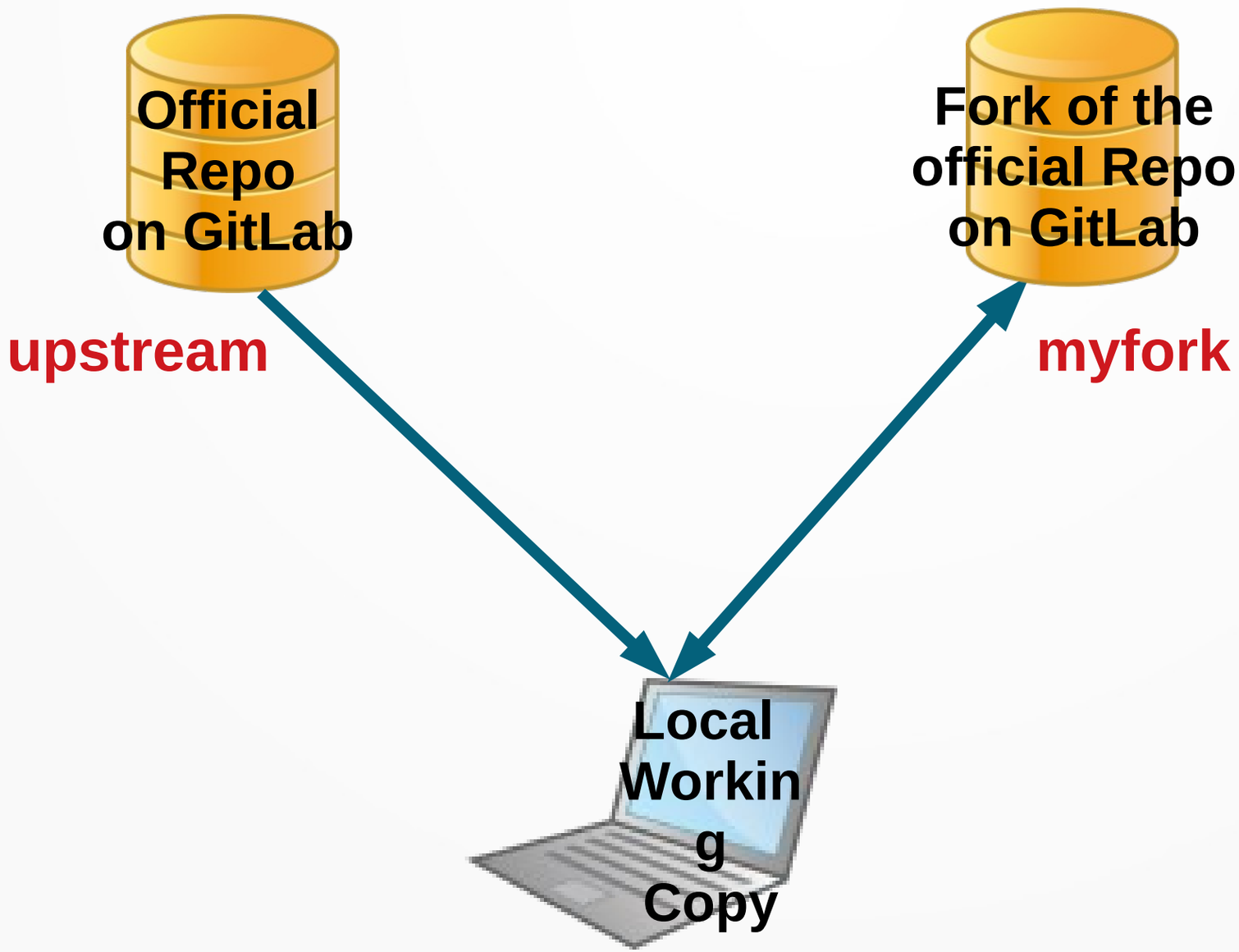


Connecting the second Repository

- You can have many remote repositories connected to your local working copy
- Local copy cloned from official repository
  - **git remote add myfork https://gitlab.in2p3.fr/<user>/base_project**

    myfork is only a name which indicates my repository fork
  - **git remote rename origin upstream**
- Local copy cloned from your fork
  - **git remote add upstream https://gitlab.in2p3.fr/f.uhlig/base_project**
  - **git remote rename origin myfork**

Connecting the Second Repository



Task

- Add your name in main.cpp such that it printed when executing the program greetings
  - Create a new branch locally
  - Add the file locally
  - Commit the changes to the new branch
  - Push the changes (Branch) to your fork

```
int main(int argc, char *argv[]){
   hello("Tesuser");
   hello(“Florian Uhlig”);
   return 0;
}
```

Upload local changes to GitLab

- In the following I assume that you know how to work with git locally
  - **git checkout -b add_my_name**
  - **git add <file>**
  - **git commit**
- How to get your changes to your fork
  - **git push <repository name> <local branch>:<repository branch>**
  - **git push myfork add_my_name:add_my_name**
    - Will upload the local branch **add_my_name** to your fork on GitLab
- Assumes that you chose the same names as I used

Workflow

Official GIT Repo

#5

#4

#3

#2

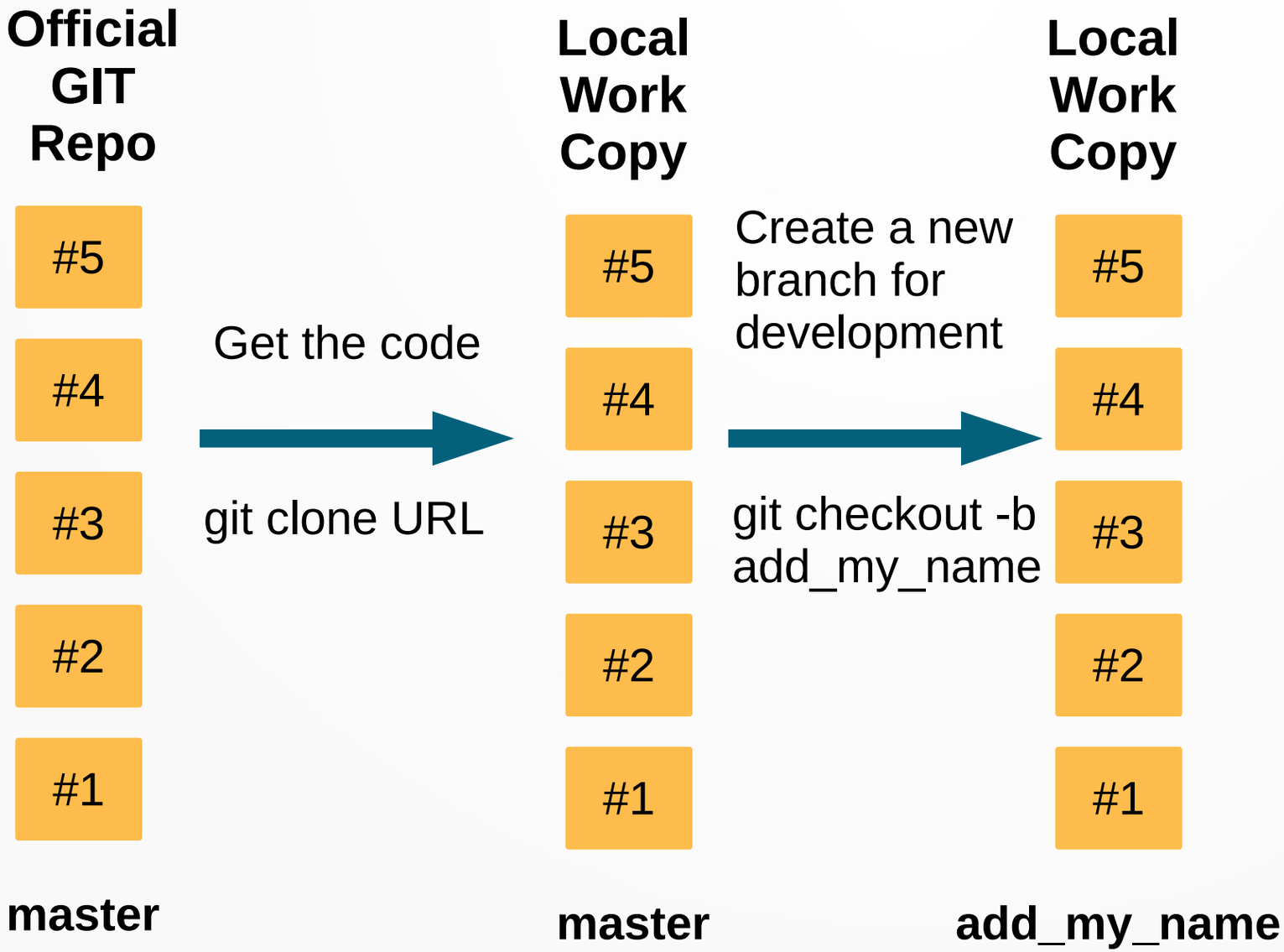
#1

master

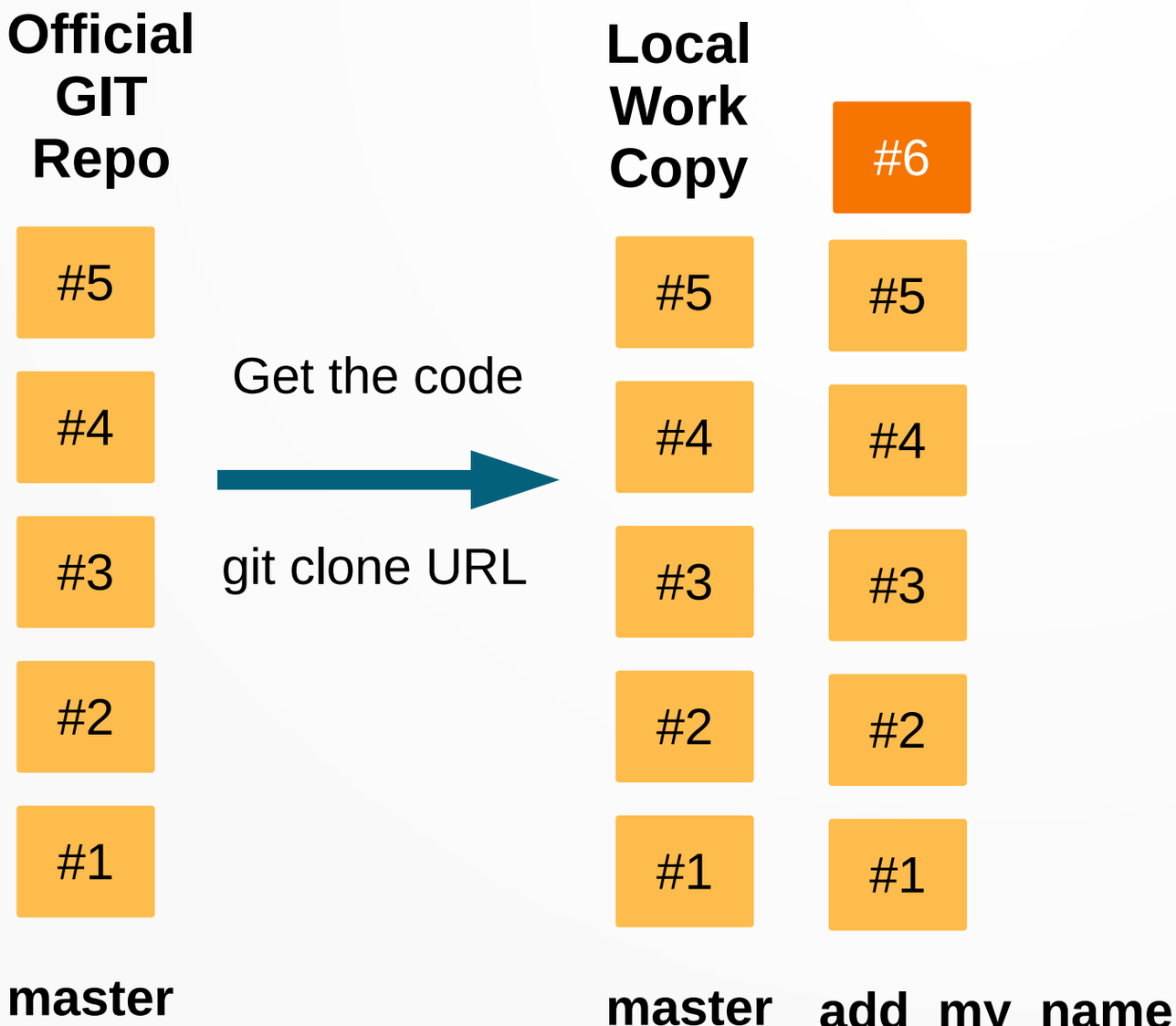
Workflow


Official GIT Repo
#5
#4
#3
#2
#1
master
Get the code
git clone URL
Local Work Copy
#5
#4
#3
#2
#1
master

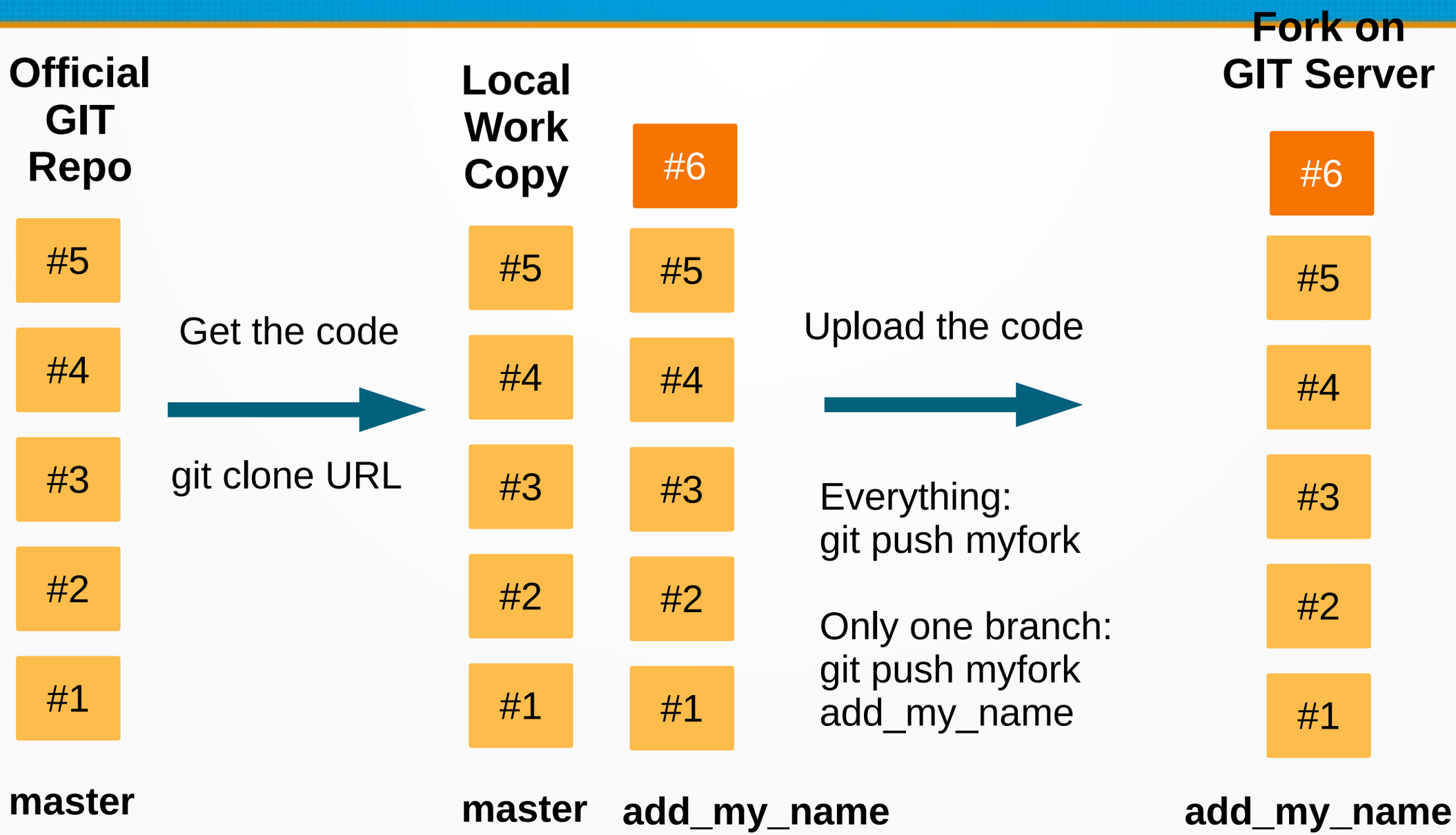
Workflow


Official GIT Repo
#5
#4
#3
#2
#1
master
Get the code
git clone URL
Local Work Copy
#5
#4
#3
#2
#1
master
Create a new branch for development
git checkout -b add_my_name
Local Work Copy
#5
#4
#3
#2
#1
add_my_name

Workflow


Official GIT Repo
#5
#4
#3
#2
#1
master
Get the code
git clone URL
Local Work Copy
#5
#4
#3
#2
#1
master
#6
#5
#4
#3
#2
#1
add_my_name

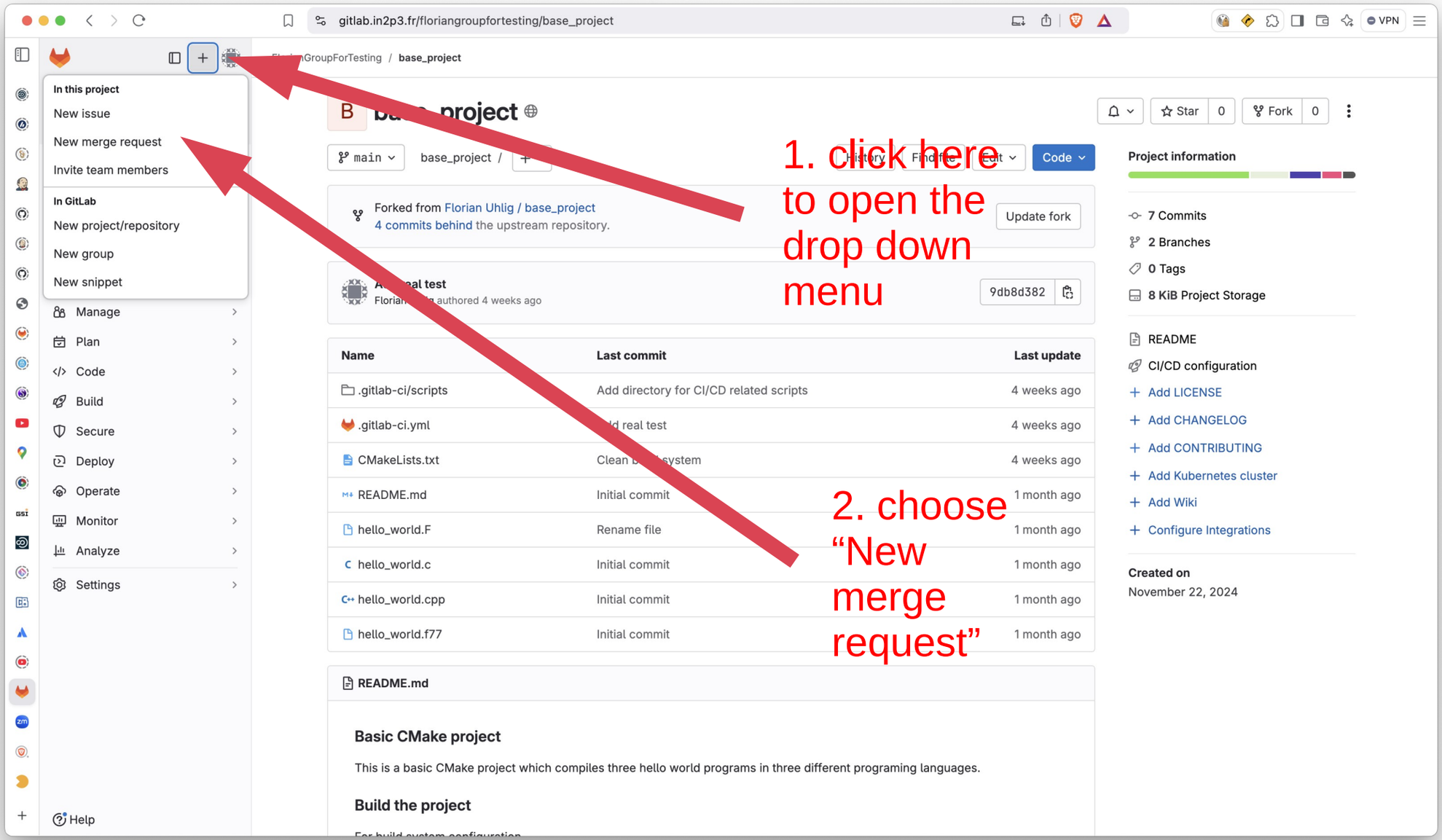
Workflow


Official GIT Repo
#5
#4
#3
#2
#1
master
Get the code
git clone URL
Local Work Copy
#5
#4
#3
#2
#1
master
#6
#5
#4
#3
#2
#1
add_my_name
Upload the code
Everything:
git push myfork
Only one branch:
git push myfork
add_my_name
Fork on GIT Server
#6
#5
#4
#3
#2
#1
add_my_name

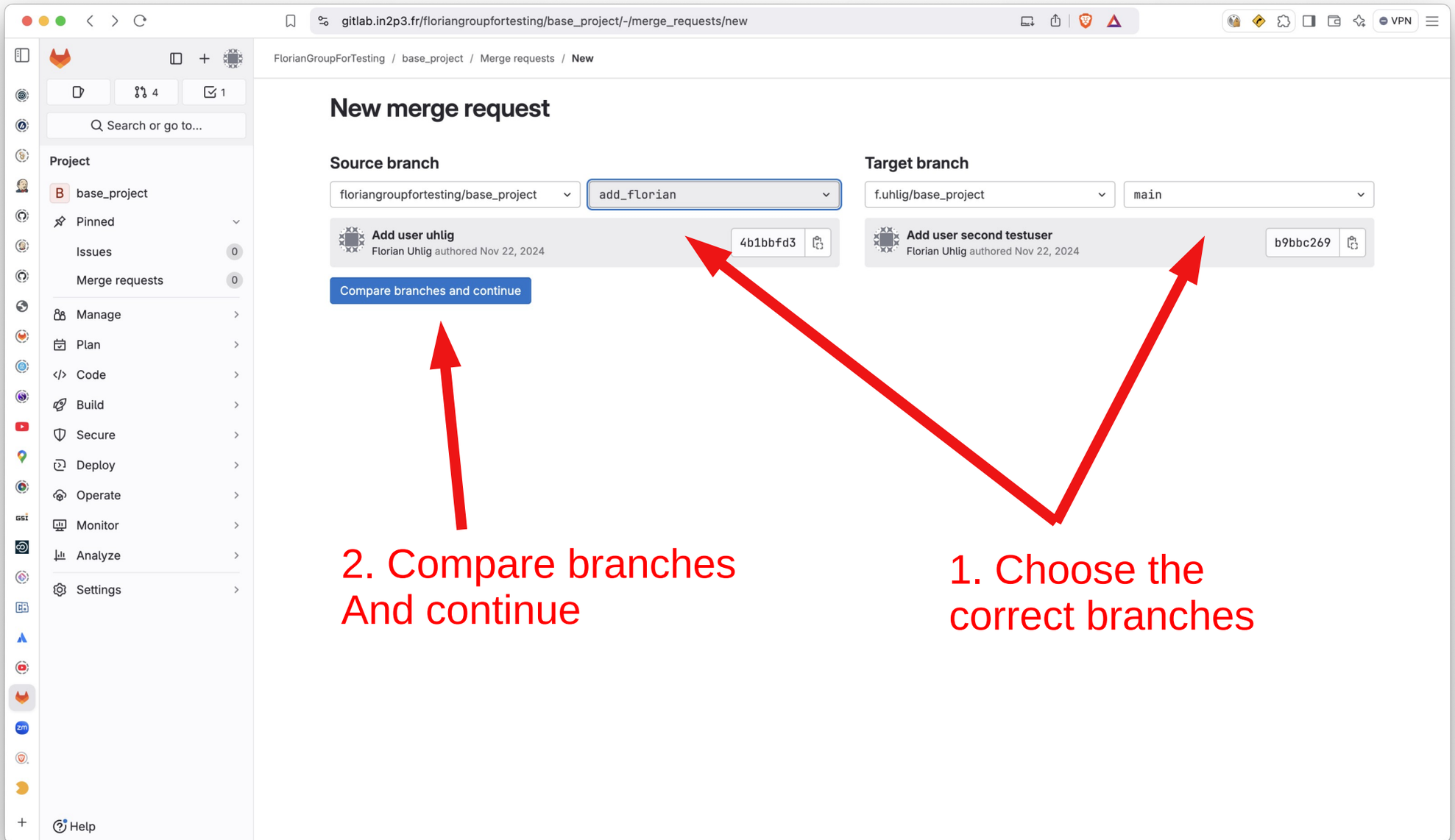
Create A Merge Request

- Navigate to your fork on the GIT server
  - For me the URL is https://gitlab.in2p3.fr/floriangroupfortesting/base_project
  - Click on “New merge request” from the “new” menu
  - Choose the proper source and target branches
    - The source branch in our example is add_may_name from my fork of base_project
    - The target branch is always main from f.uhlig/base_project
  - Click on compare branches and continue
    - If everything is correct you can submit the merge request
    - Problems are discussed later
  - Now we come to the screenshots

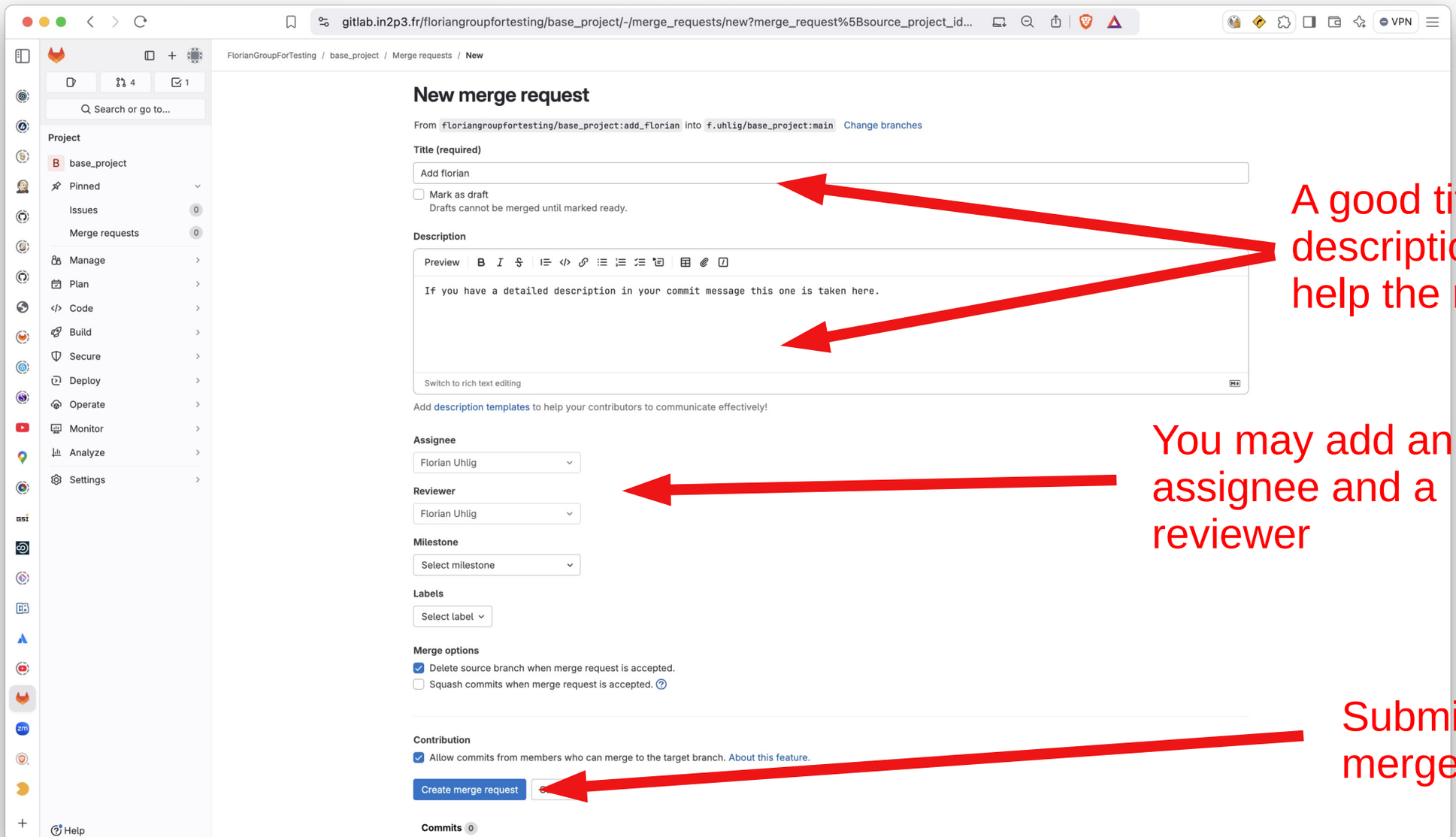
Merge Request



Merge Request



Merge Request



Merge Request Feedback

- After the merge request (MR) is submitted there are currently some checks (Continuous Integration) done
  1) Check if the MR is properly rebased
  2) Check if the history is linear
  3) Reviewer will check the code and give feedback and may request changes (not automatic)
- All above stages may fail which will need updates from you
- Do the changes and push the changes to the same branch
  - The update of the branch will trigger steps 1-3 again
  - Repeat until the merge request is accepted and merged
  - Relax

CI checks in detail

- All checks are defined in the file .gitlab-ci.yml
- Checks can be separated in stages e.g. build and test
- Several test per stage are possible

```
stages:
 - checkRepository

RebaseCheck:
 stage: checkRepository
 variables:
  GIT_DEPTH: 200
 image: alpine
 only:
  refs:
   - merge_requests
  variables:
   - $CI_MERGE_REQUEST_PROJECT_PATH == "f.uhlig/base_project" && $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "main"
 Script:
    - echo “Hello”
```

docker image (alpine) used for the check

Run only if a merge request to main branch of project f.uhlig/base_project

Script/Program doing the actual check

Rebase check

- Get the commit hash of the main branch of the upstream repository (HEAD of the upstream repository)
  - **git show-ref upstream/main | cut -f1 -d' '**
- Get the commit hash of the commit where the MR branch was branched off the upstream/main branch
  - **git merge-base upstream/main HEAD**
- If both commits are equal the MR branch was properly rebased

Linear history check

- Check if there are any merge commits in the commits of the MR branch
  - **git rev-list --min-parents=2 --count upstream/main..HEAD**
- git rev-list
  - Get the list of commits
- --min-parents=2 –count
  - Filter the list for commits with two parents (merge commits) and count them
  - Rebase commits have only one parent -> linear history
- upstream/main..HEAD
  - Use only commits which will be added with the current merge request

Merge conflicts

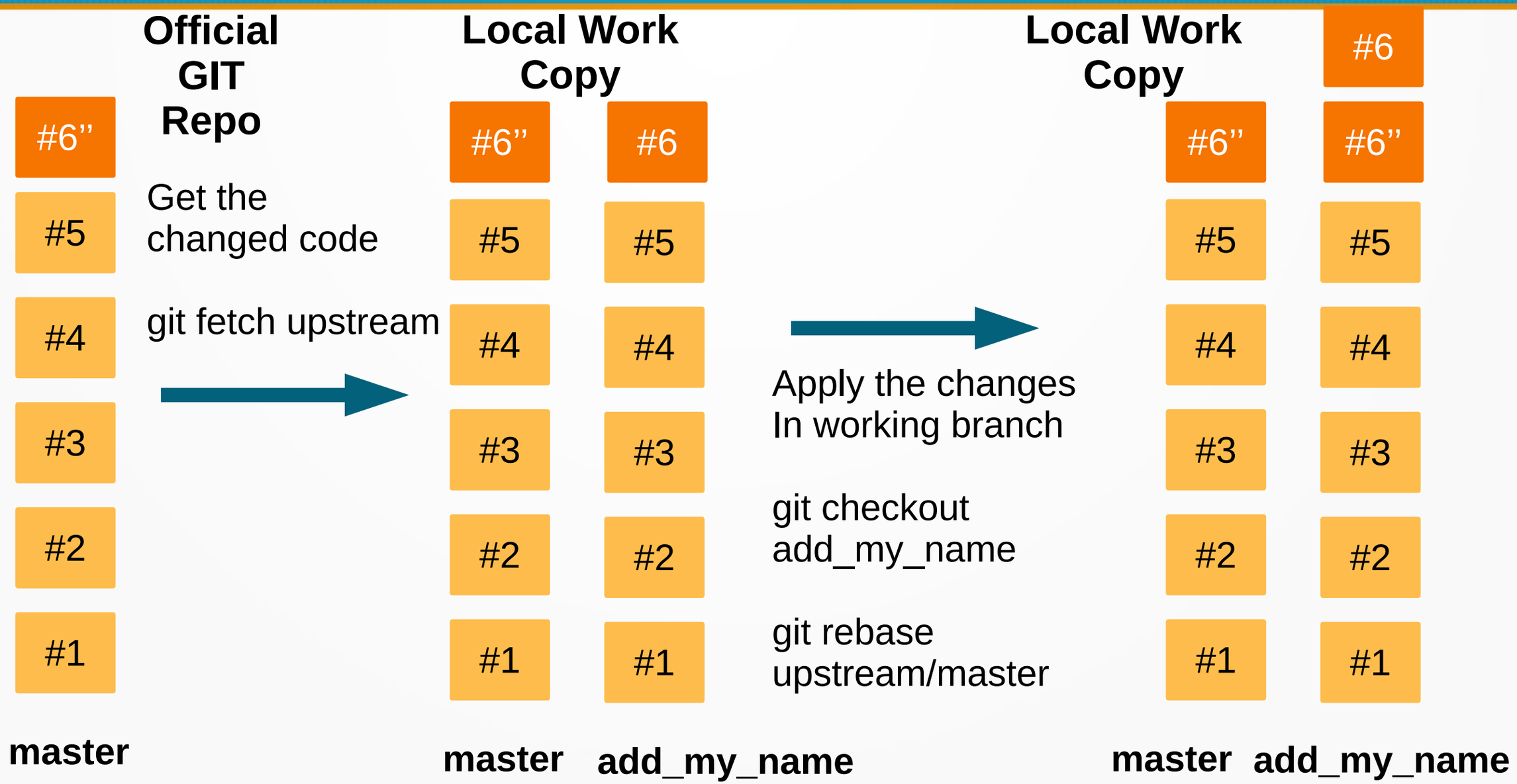
- In reality not very frequent
- Since many users will change the same file and probably the same line most of you will have to fix merge conflicts
  - Nothing to worry about
  - In most cases GIT can resolve the conflicts for you
  - Conflicts are clearly shown in the files such that you can solve them manually

Merge Request Conflict

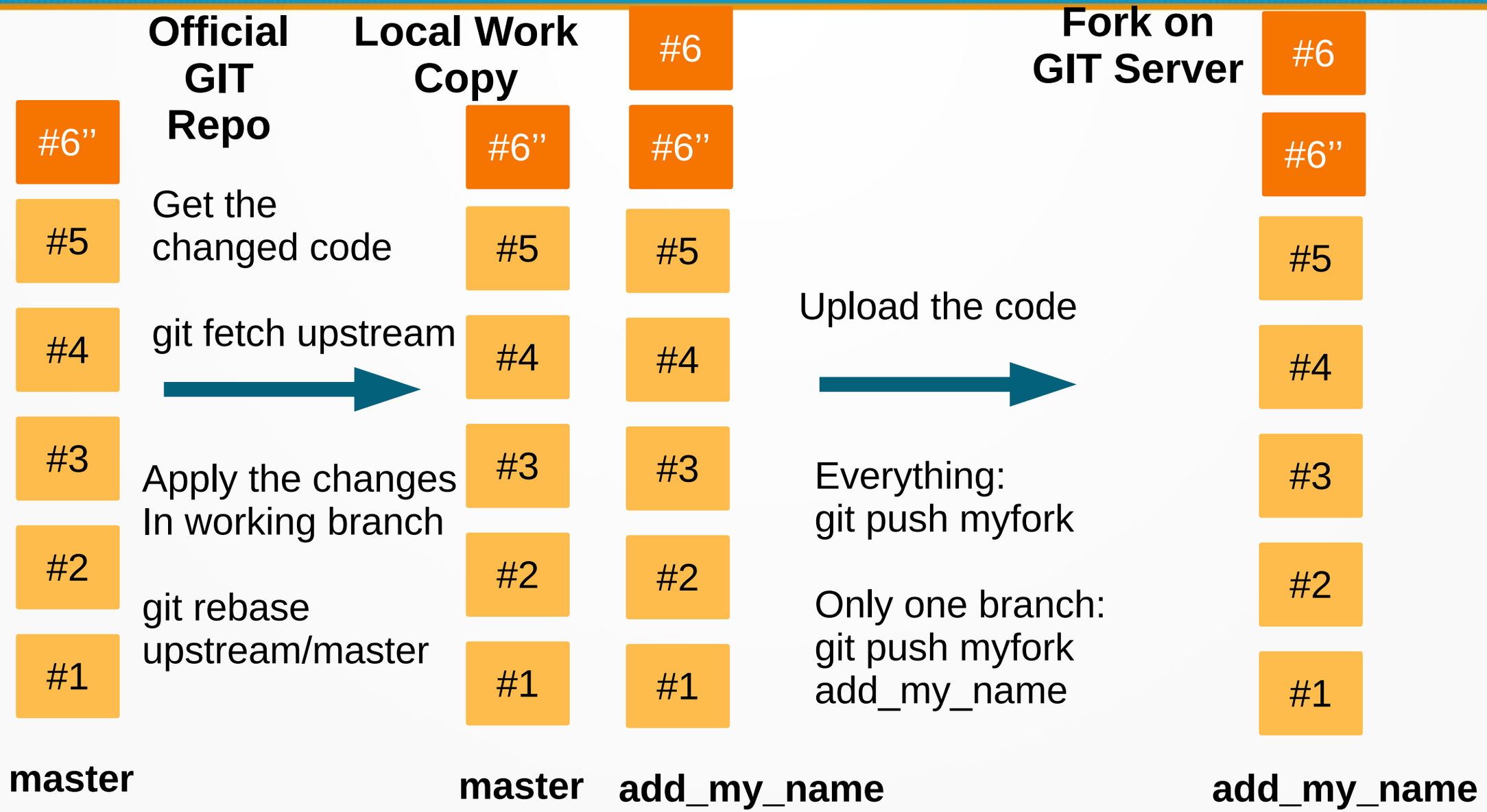
Merge Request Conflicts

- Depending on the settings of the fork it may be possible to fix the conflict on the server
  - Can be done by an admin of the official project
  - No user intervention needed
  - Will change code in your fork !!!
    - Always use an extra branch which will be deleted after merging
- Please remember to always do a “rebase” before you create a merge request
  - Start the merge request with a clean state

Solve the conflict locally



Push the changes



Weak point of rebase workflow

- Since you changed the local history GitLab will not accept the push
  - GitLab detects history mismatch
- Compare your local branch to the remote branch on the server to be sure that you only commit the wanted changes
  - **git diff myfork/add_my_name**
- If the diff only shows the expected changes do a force push
  - **git push myfork add_my_name –force-with-lease**
- If you are unsure create a new branch and a new merge request on the servers
  - **git push myfork add_my_name:add_my_name_2**
  - Don’t forget to cleanup after your MR was merged

Conclusion

- Hope you got an idea about GIT and the proposed workflow
  - Ask questions, discuss or complain
  - I am around till Wednesday evening to sort out problems
  - Don’t be afraid to use git, if you are unsure you always can create another branch for testing
- Get a free backup when using a remote GitLab/GitHub server
- Use GIT for everything where you want to keep the history of development
  - Code
  - Thesis
  - Paper