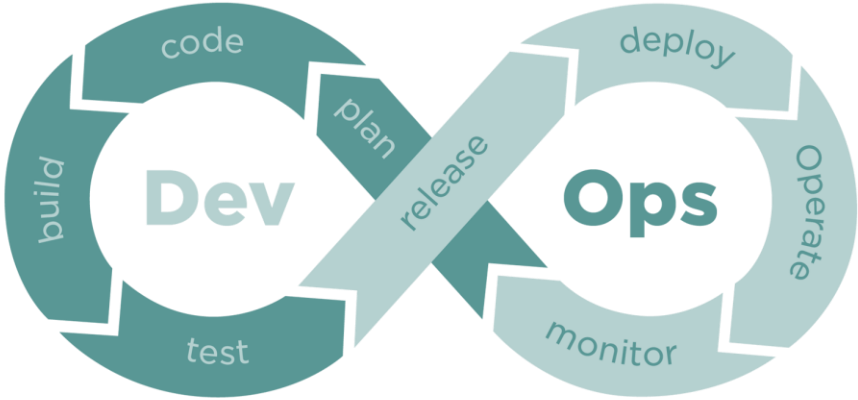
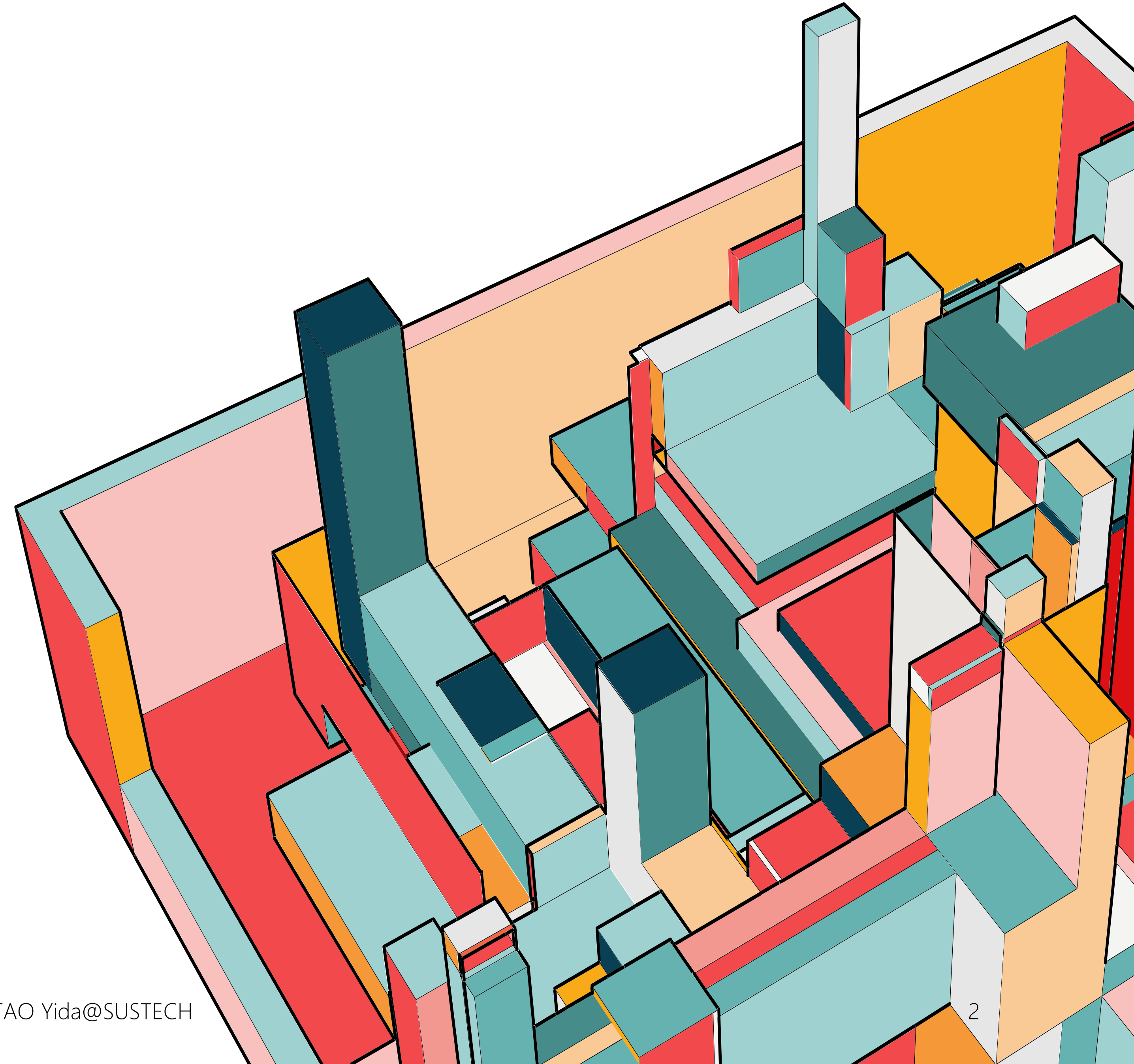
CS304 SOFTWARE ENGINEERING

Yida Tao
taoyd@sustech.edu.cn

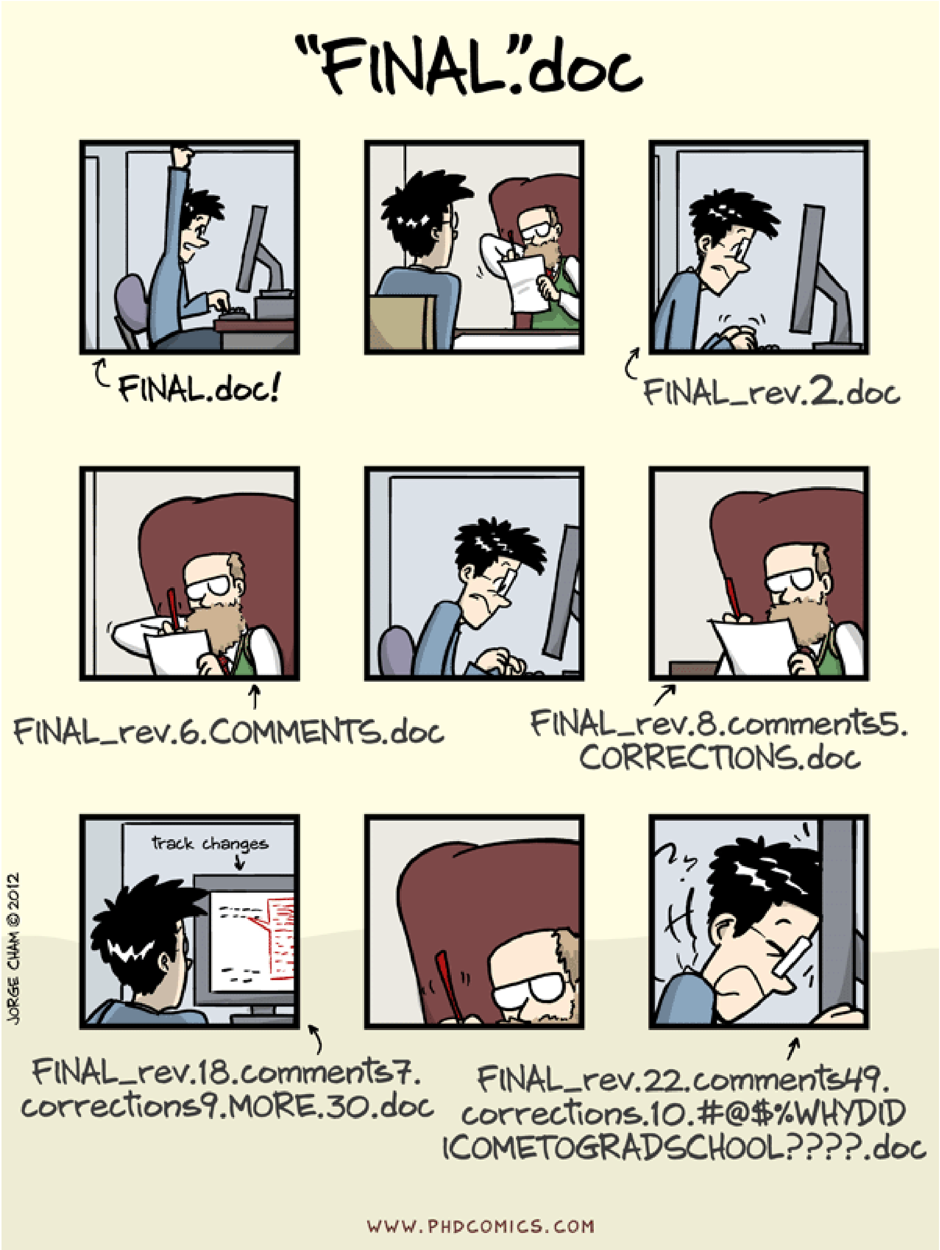
LECTURE 3

- Version Control
- Git





WHY VERSION CONTROL?



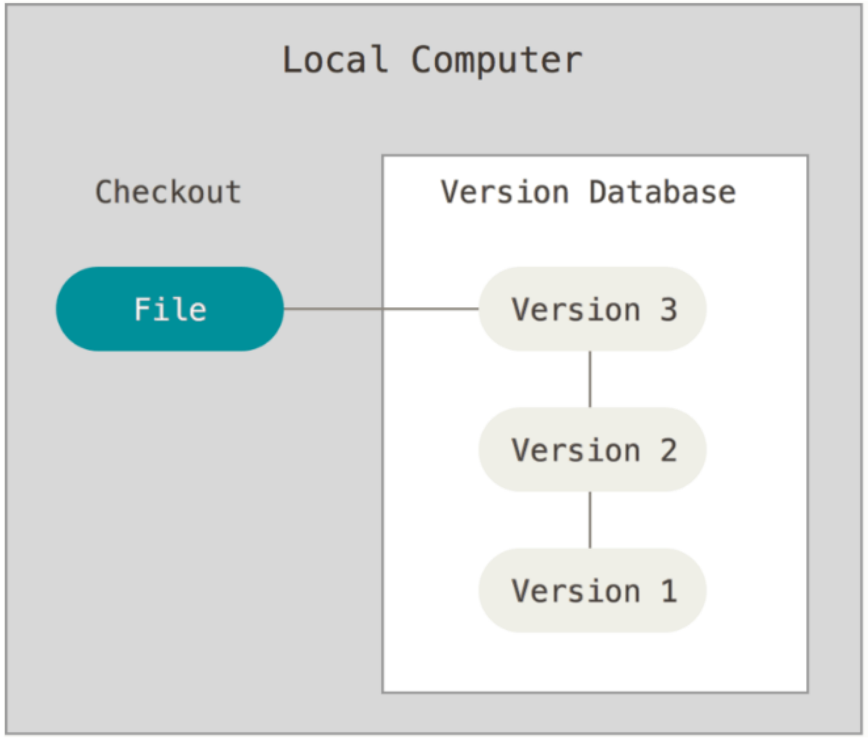
WHAT IS VERSION CONTROL?

- **Version control** is the practice of tracking and managing changes to software code and other software artifacts like documentations
- **Version control systems (VCS)** are software tools that help software teams track and manage modifications to files (typically source code) over time.
    - Every modification is stored in a special kind of database
    - We can **revert** selected files back to a previous state, revert the entire project back to a previous state, **compare** changes over time, see **who** last modified something that might be causing a problem, who introduced an issue and when, and more.
- VCS are especially useful for DevOps teams since they help them to **reduce risks** caused by continuous changes and **increase successful deployments**.

https://www.atlassian.com/git/tutorials/what-is-version-control

LOCAL VERSION CONTROL SYSTEMS

- Local VCSs have a simple database that kept all the changes to files under revision control.
- One of the most popular VCS tools was a system called RCS
- RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

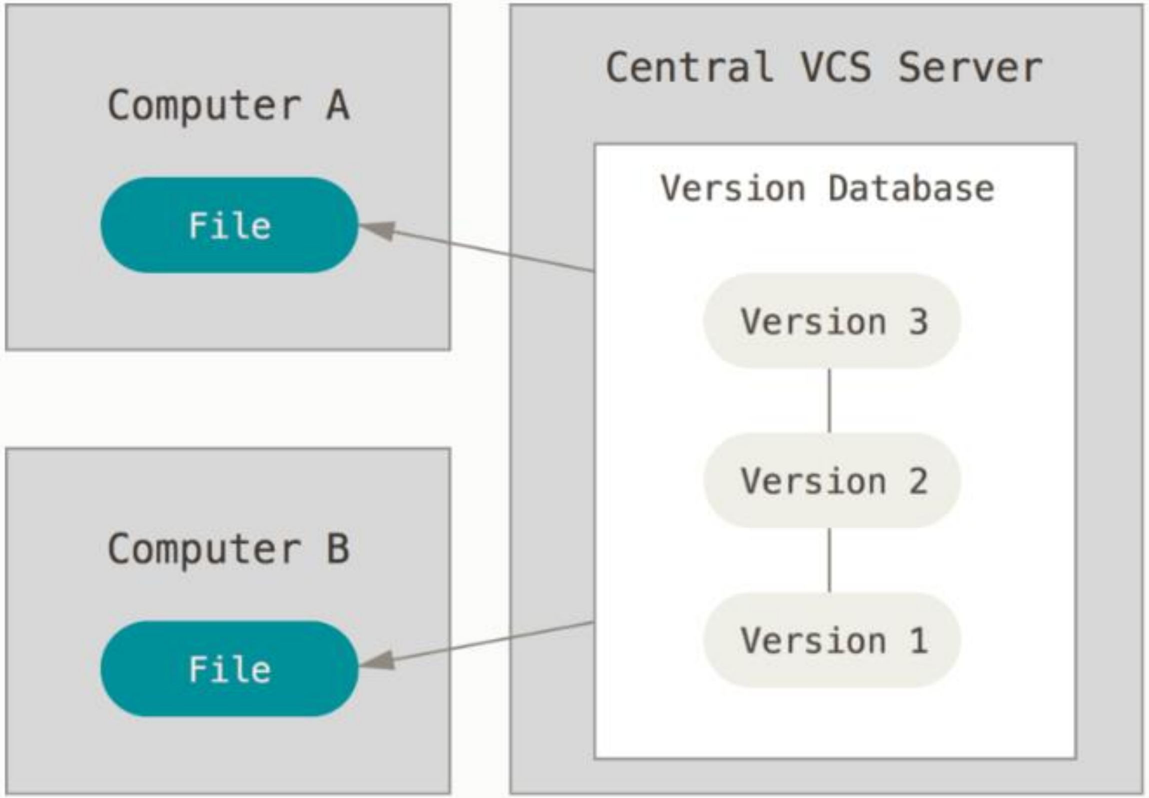


https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

Problems?

CENTRALIZED VERSION CONTROL SYSTEMS

- Centralized VCS emerged since developers need to collaborate with each other
- Centralized VCS (e.g., CVS, Subversion/SVN, and Perforce) have **a single server** that contains all the versioned files, and several clients that **check out** files from that central place.
- Using CVCS, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what.



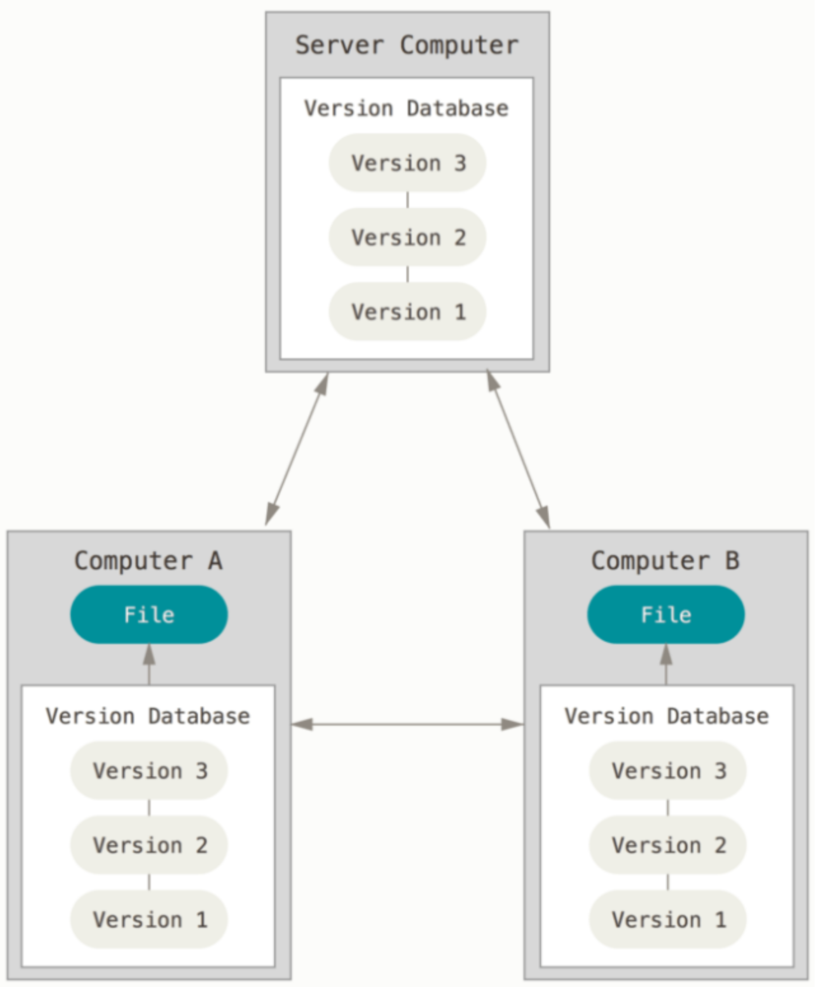
https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control

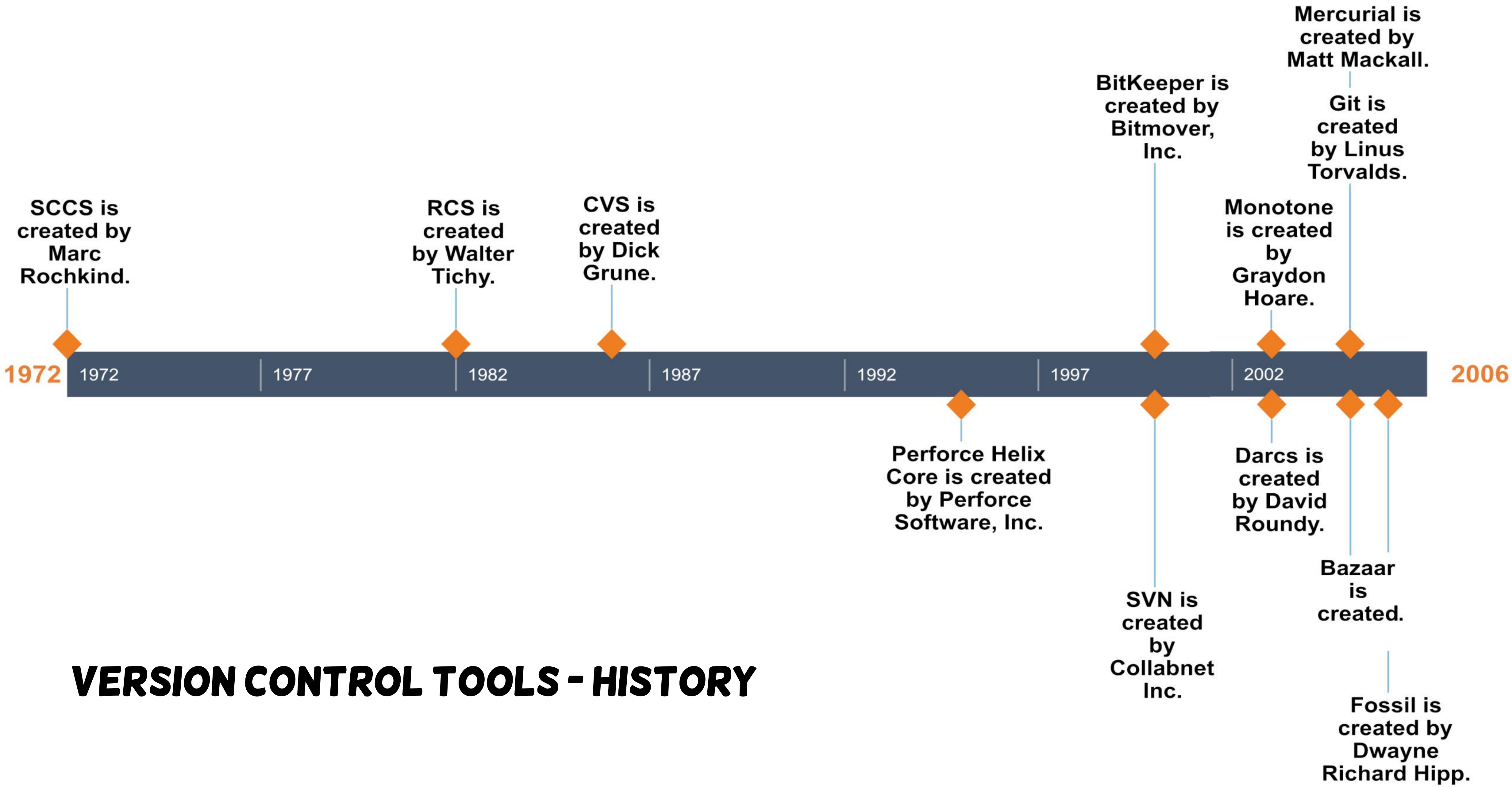
Problems?

DISTRIBUTED VERSION CONTROL SYSTEMS

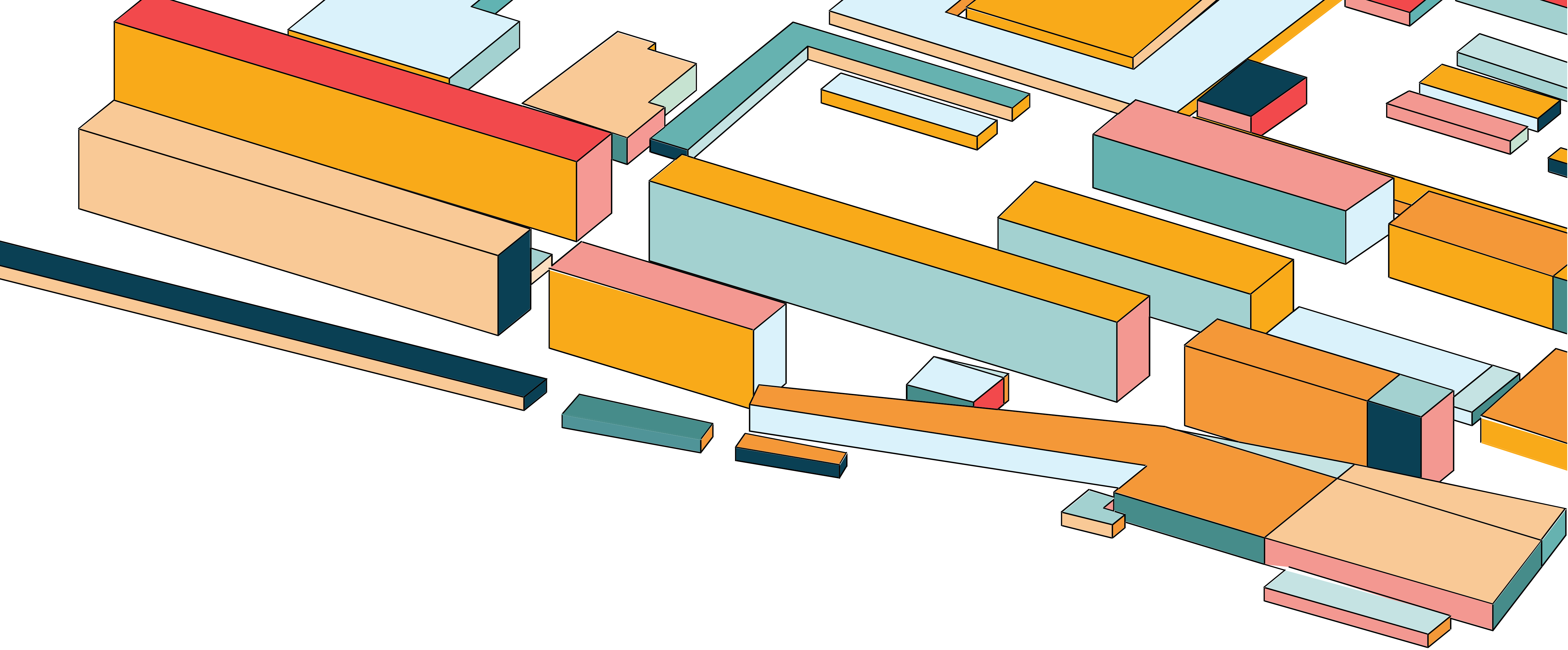
- In a Distributed VCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files
- Rather, they fully mirror the repository, including its full history. Every clone is really a full backup of all the data.
- Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it.

https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control





VERSION CONTROL TOOLS - HISTORY



GIT BASICS

WHAT IS GIT?

Snapshots, Not Differences

Git stores stream of snapshots, while other VCS stores delta.

Nearly Every Operation Is Local

Most operations in Git need only local files and resources to operate.

Git Has Integrity

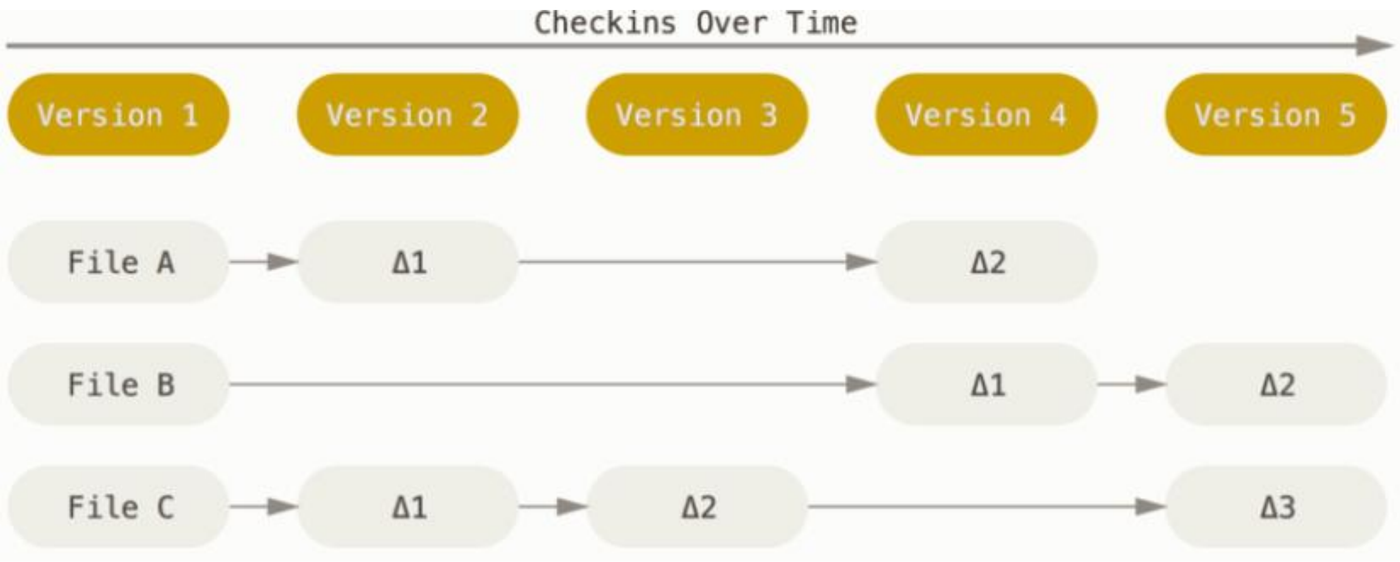
Everything in Git is checksummed before it is stored and is then referred to by that checksum.

Git Generally Only Adds Data

After you commit a snapshot into Git, it is very difficult to lose the data.

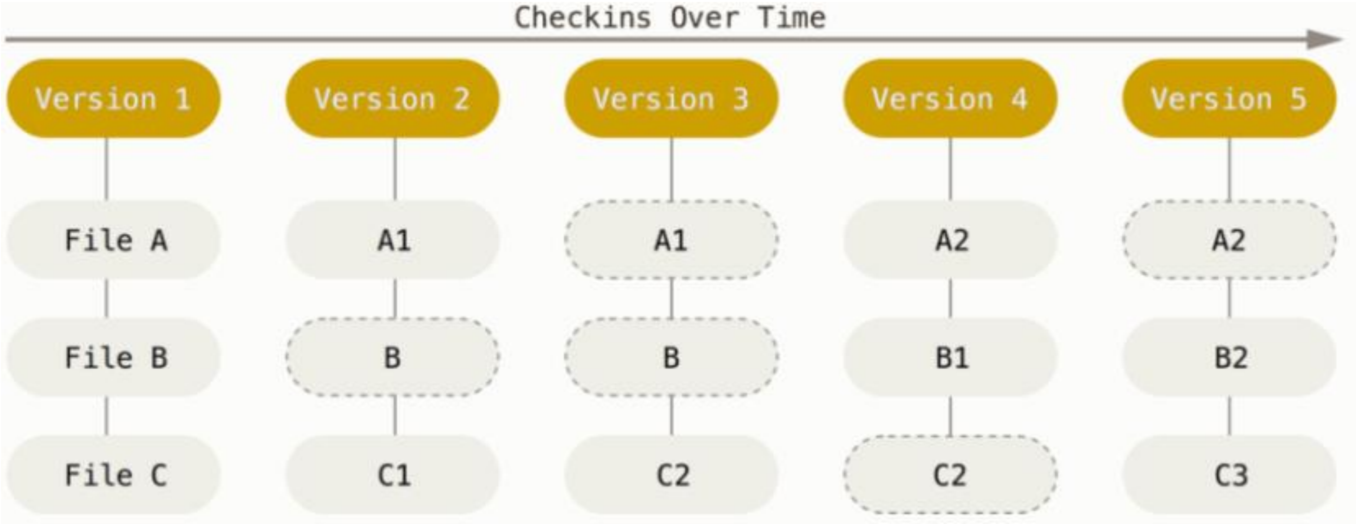
SNAPSHOTS, NOT DIFFERENCES

- The major difference between Git and any other VCS is the way Git thinks about its data.
- Conceptually, most other systems store information as a list of file-based changes.
- These other systems (e.g., CVS, SVN, Perforce) think of the information they store as a set of files and the changes made to each file over time (this is commonly described as **delta-based version control**).



SNAPSHOTS, NOT DIFFERENCES

- Git thinks of its data more like a series of snapshots of a miniature filesystem.
- With Git, every time you commit, or save the state of your project, Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.
- To be efficient, if files have not changed, Git doesn't store the file again, just a link to the previous identical file it has already stored.



NEARLY EVERY OPERATION IS LOCAL

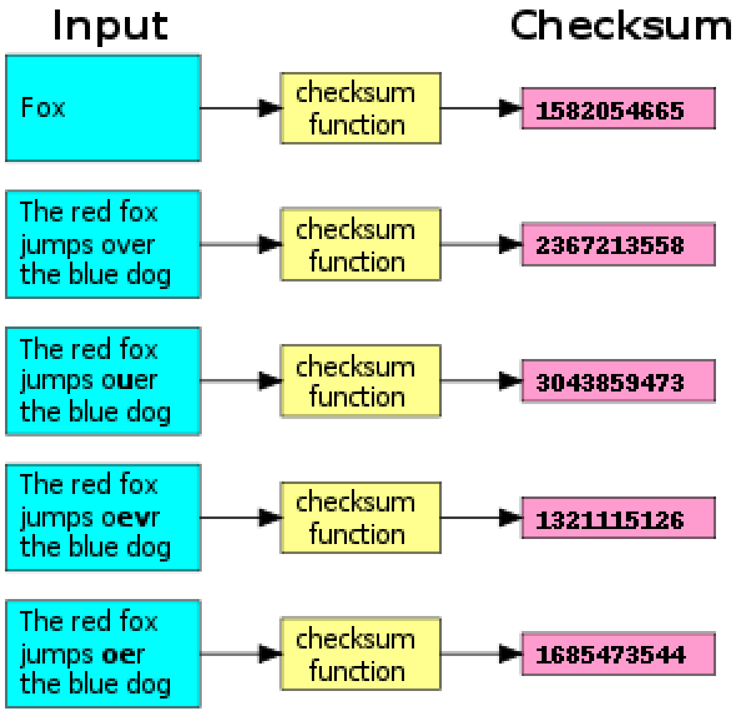
- Most operations in Git need only local files and resources to operate — generally no information is needed from another computer on your network, meaning that there is very little you can't do if you're offline or off VPN

- For example, to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you — it simply reads it directly from your local database.

- If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally.

GIT HAS INTEGRITY

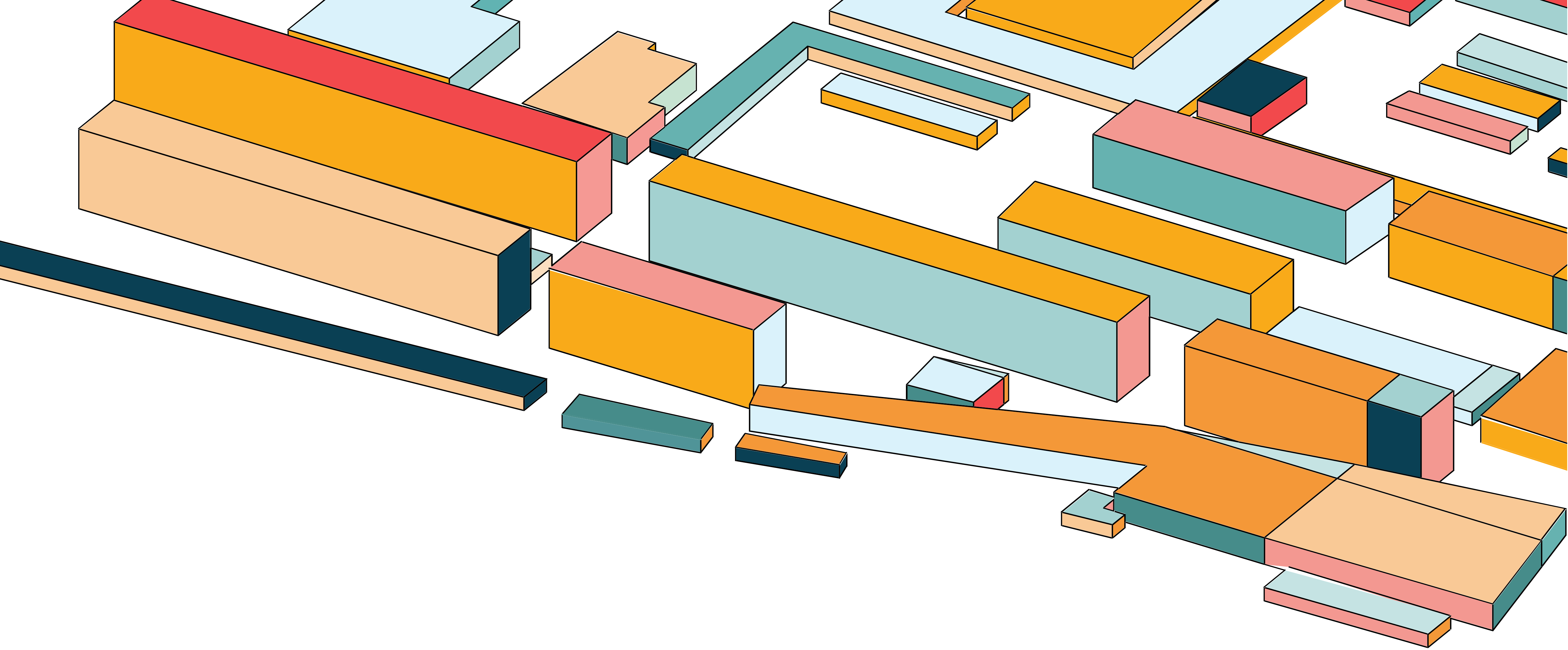
- Everything in Git is checksummed before it is stored and is then referred to by that checksum.
- It's impossible to change the contents of any file or directory without Git knowing about it.
- The mechanism that Git uses for this checksumming is called a SHA-1 hash. This is a 40-character string composed of HEX characters and calculated based on the **contents of a file** or **directory structure** in Git.
- Git stores everything in its database not by file name but by the hash value of its contents.



A checksum is a small-sized block of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage.

GIT GENERALLY ONLY ADDS DATA

- When you do actions in Git, nearly all of them only add data to the Git database.
- It is hard to get the system to erase data in any way.
- After you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.



GIT ARCHITECTURE

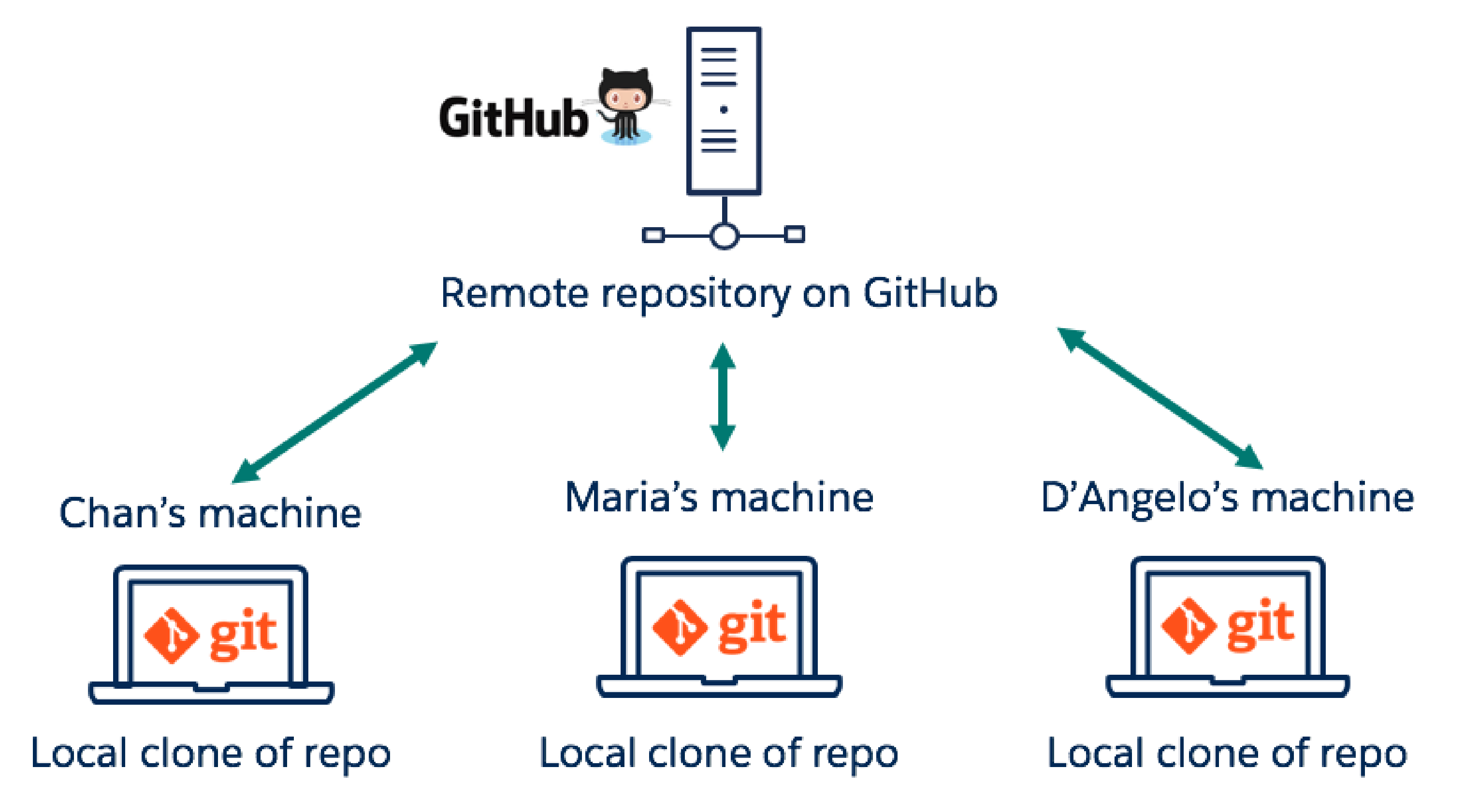
GIT LOCAL REPO VS. REMOTE REPO

Local Repo

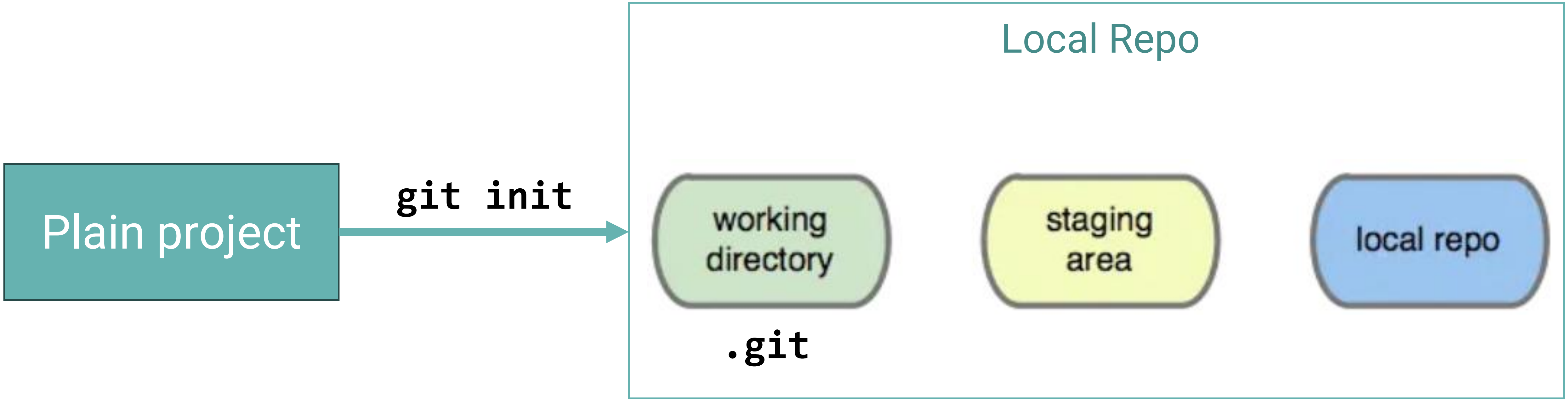
- The one on which we will make local changes, typically this local repository is on our computer.
- Users have full git features on local git repos, even without Internet access

Remote Repo

- The one on a remote server (e.g., GitHub, GitLab, BitBucket)
- The purpose of a remote repository is to host/publish your code to the world and allow people to read or write it.



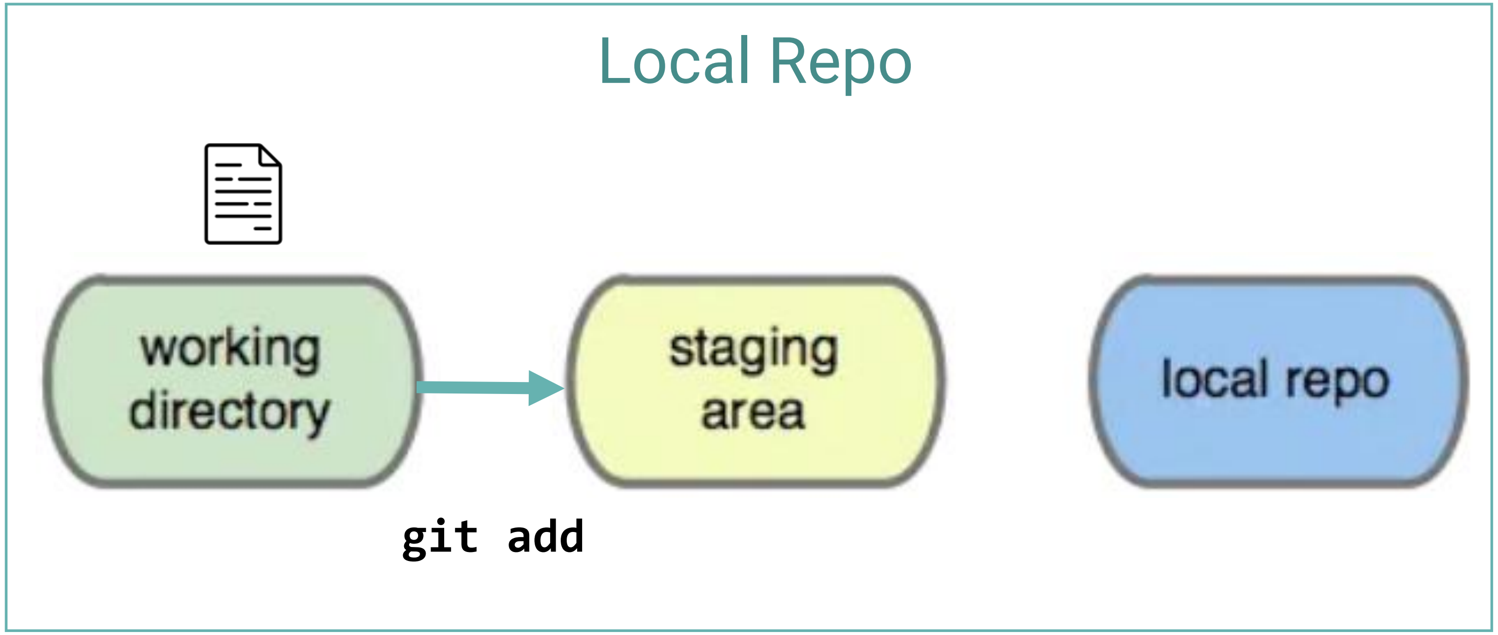
GIT LOCAL REPO : WORKFLOW



To turn a normal project folder into a git working directory, use **`git init`**

GIT LOCAL REPO : WORKFLOW

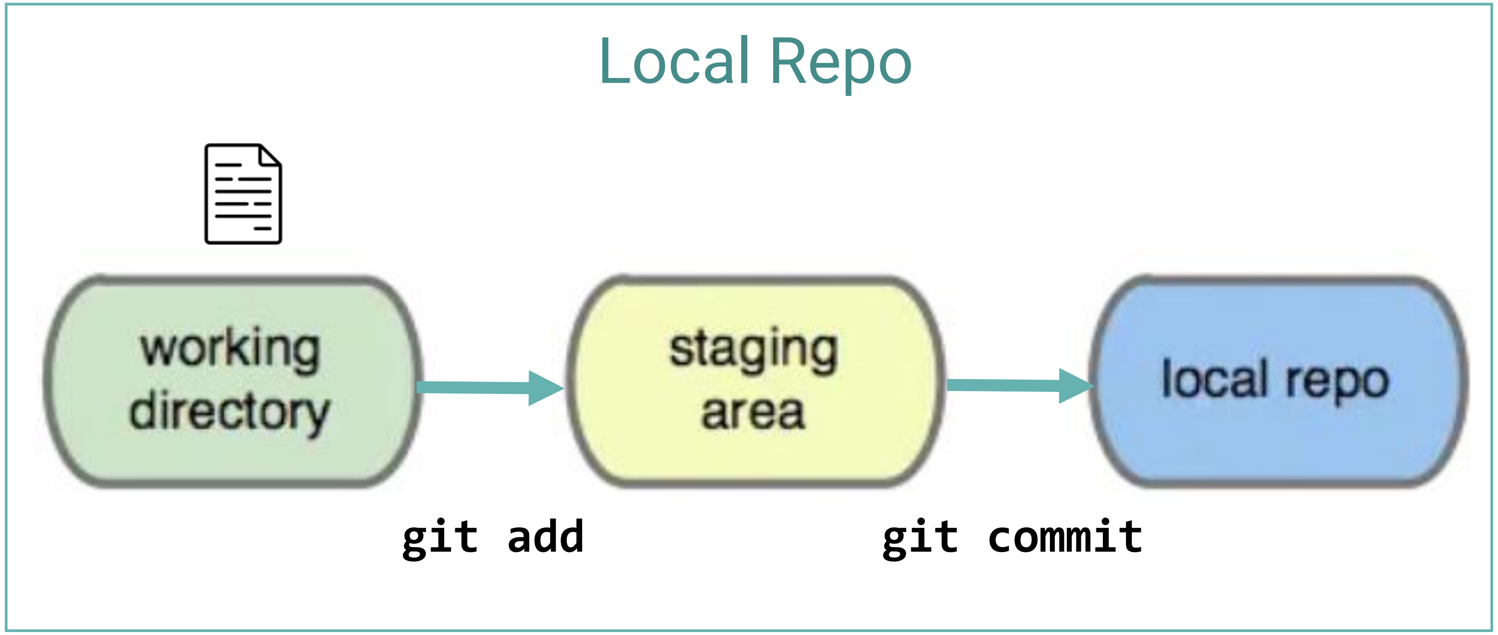
- Untracked files are the ones still NOT versioned—"tracked"—by Git. This is the state of new files you add to your repository.
- Git is aware the file exists, but still hasn't saved it in its internal database.
- If you lose the information from an untracked file, it's typically gone for good. Git can't recover it since it didn't store it in the first place.



To stage files to be part of the next commit, use `git add xxx`

GIT LOCAL REPO : WORKFLOW

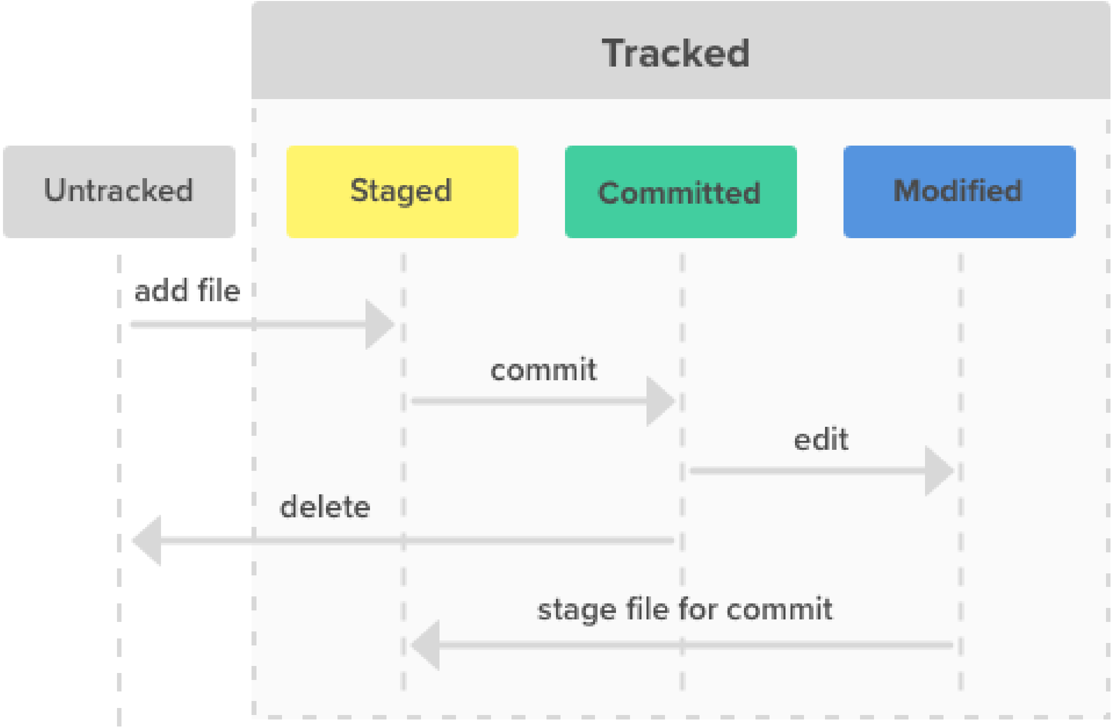
- **Staging area (index)**: a file in your Git directory, that stores information about what will go into your next commit.
- **Local repo**: where Git stores the metadata and object database for your project.



To take the files as they are in the staging area and store that snapshot permanently to your local git directory, use `git commit`

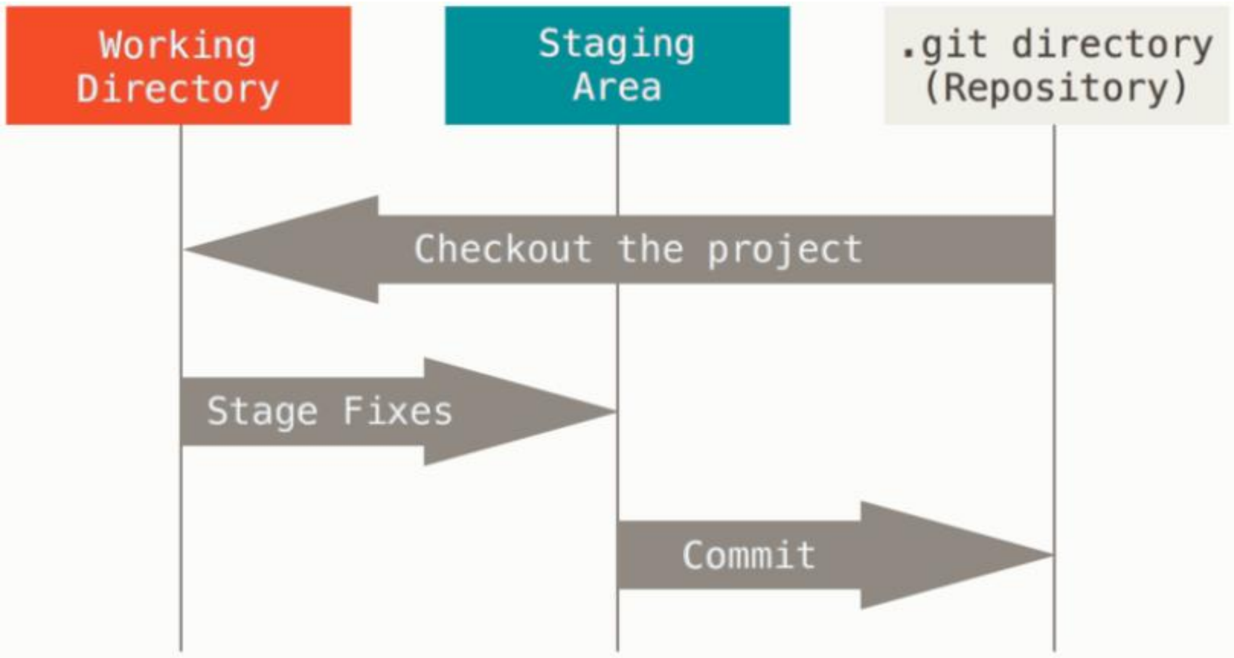
GIT LOCAL REPO - FILE STATES

- **Untracked:** files not tracked by Git
- **Staged**: means that you have marked a modified file in its current version to go into your next commit snapshot.
- **Committed**: means that the data is safely stored in your local database.
- **Modified**: means that you have changed the file but have not committed it to your database yet.



GIT LOCAL REPO – 3 SECTIONS

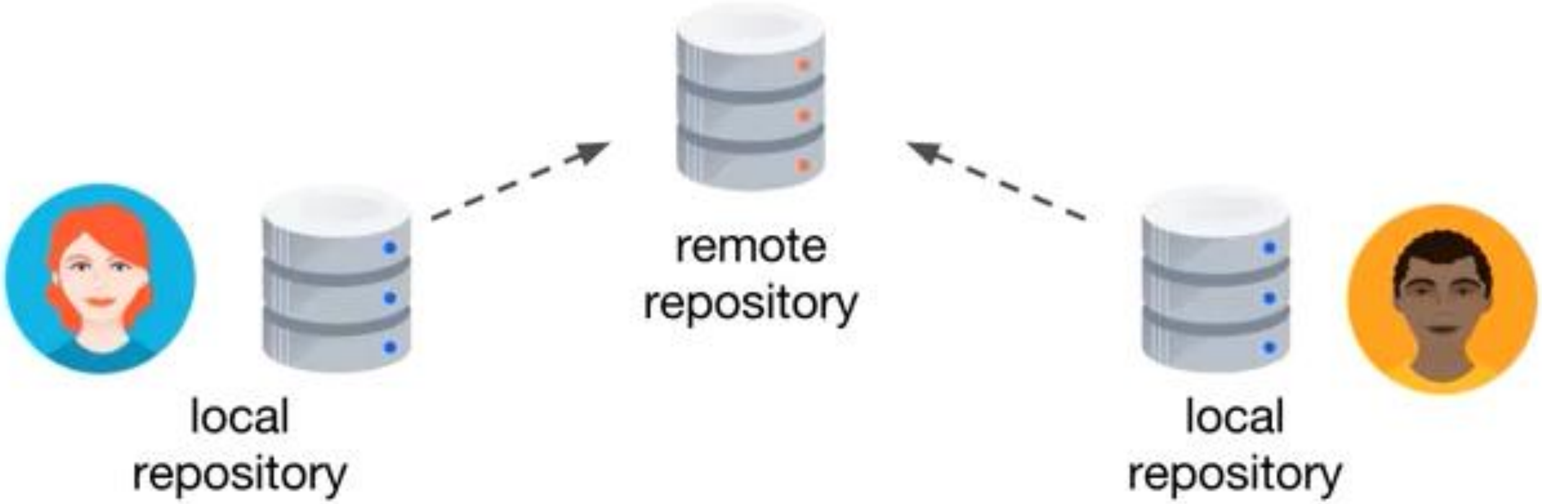
- **Working tree**: a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- **Staging area (index)**: a file in your Git directory, that stores information about what will go into your next commit.
- **Git directory**: where Git stores the metadata and object database for your project. This is what is copied when you `clone` a repository from another computer.



WORKING WITH REMOTE REPO

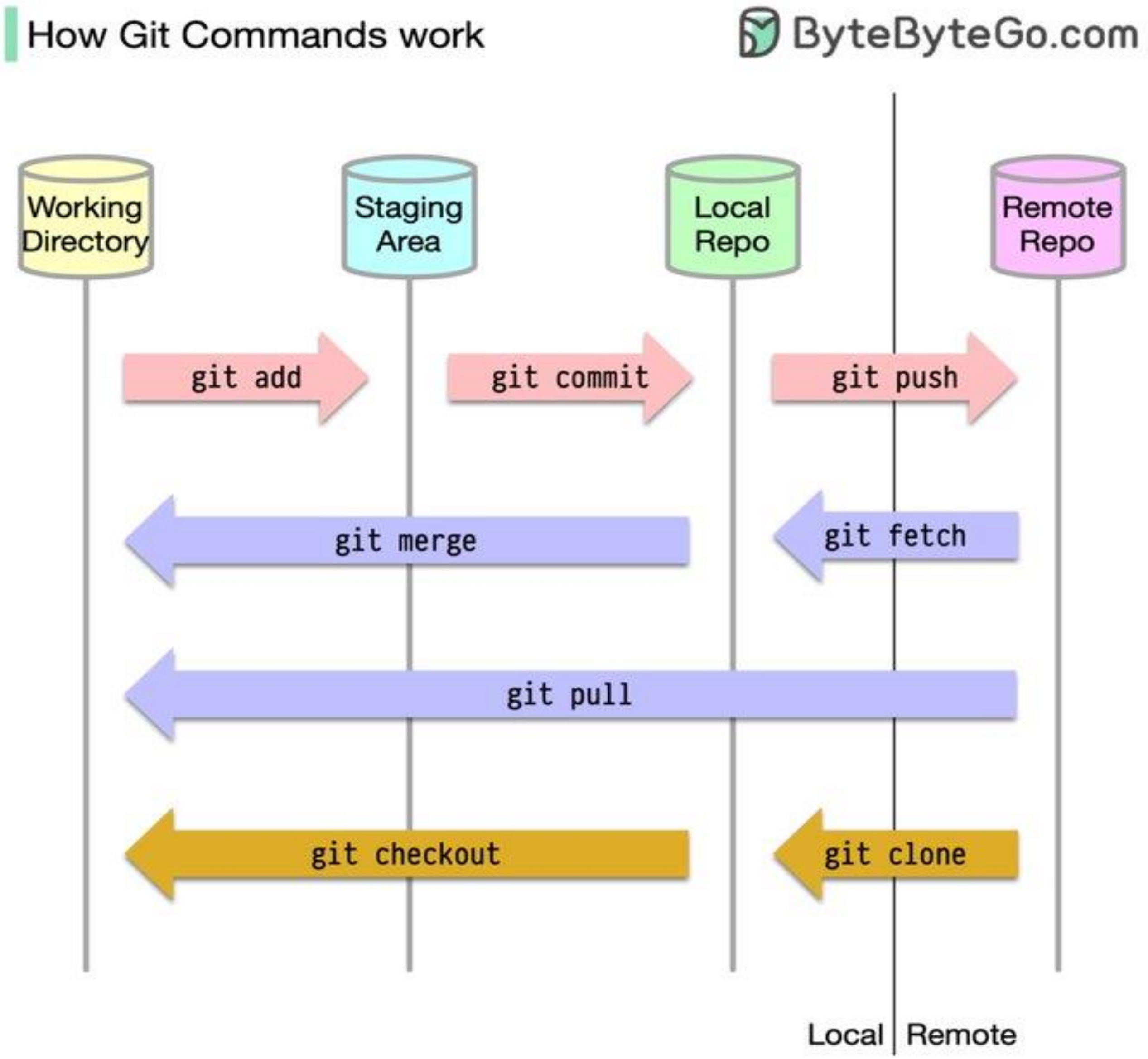
To collaborate, you can “link” a local repo and a remote repo

- Approach 1: Using `git init` and `git remote add <name> <URL>` on local repo (`origin` is the default name)
- Approach 2: Using `git clone` to clone a remote repo, which also implicitly setups the remote for you



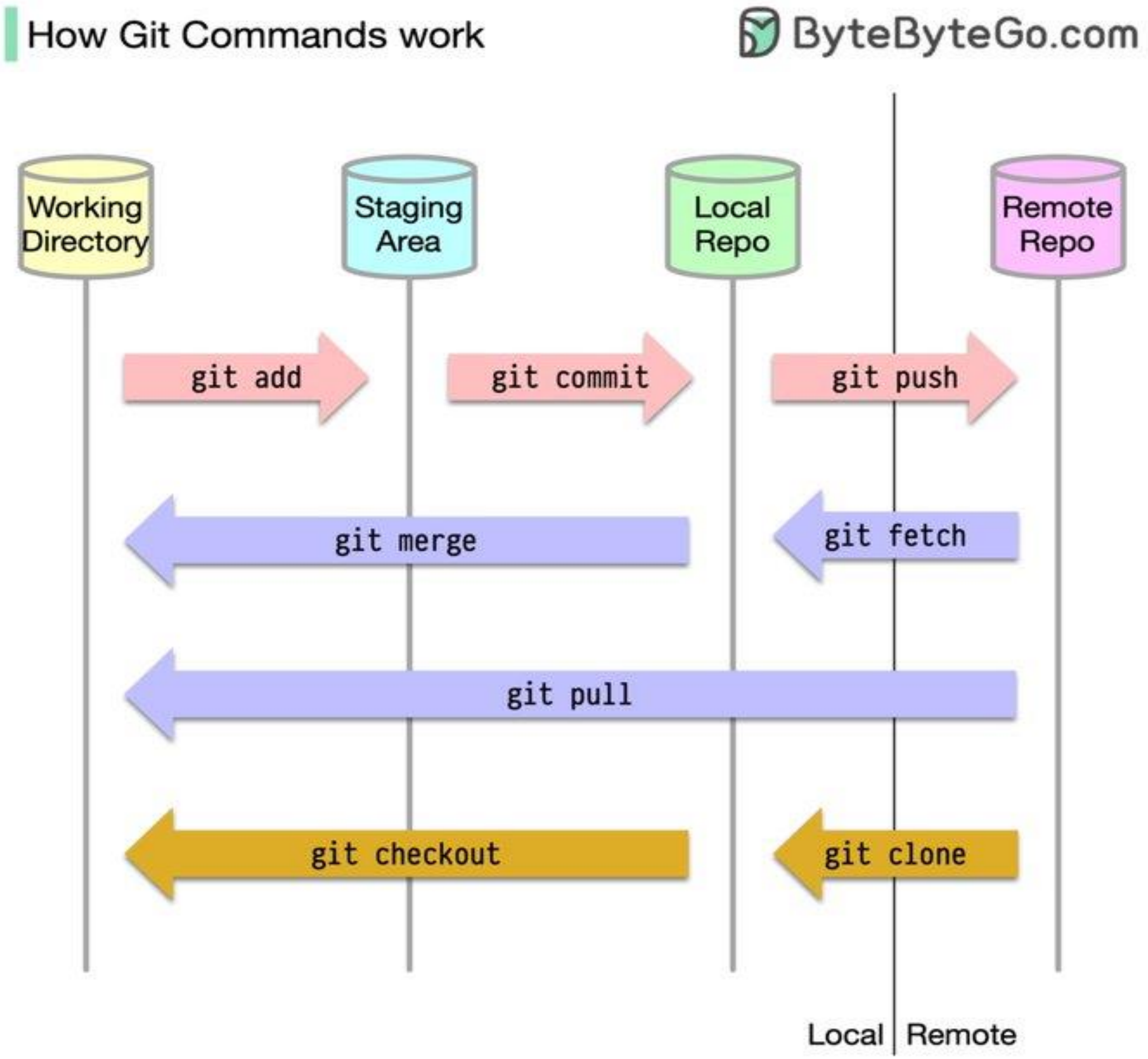
GIT CLONE

- Clones a repository into a newly created directory.
- Cloning automatically creates a remote connection called "origin" pointing back to the original repository.
- Cloning creates and checks out an initial branch from the cloned repository’s currently active branch (HEAD).
- Cloning also makes other necessary setups.



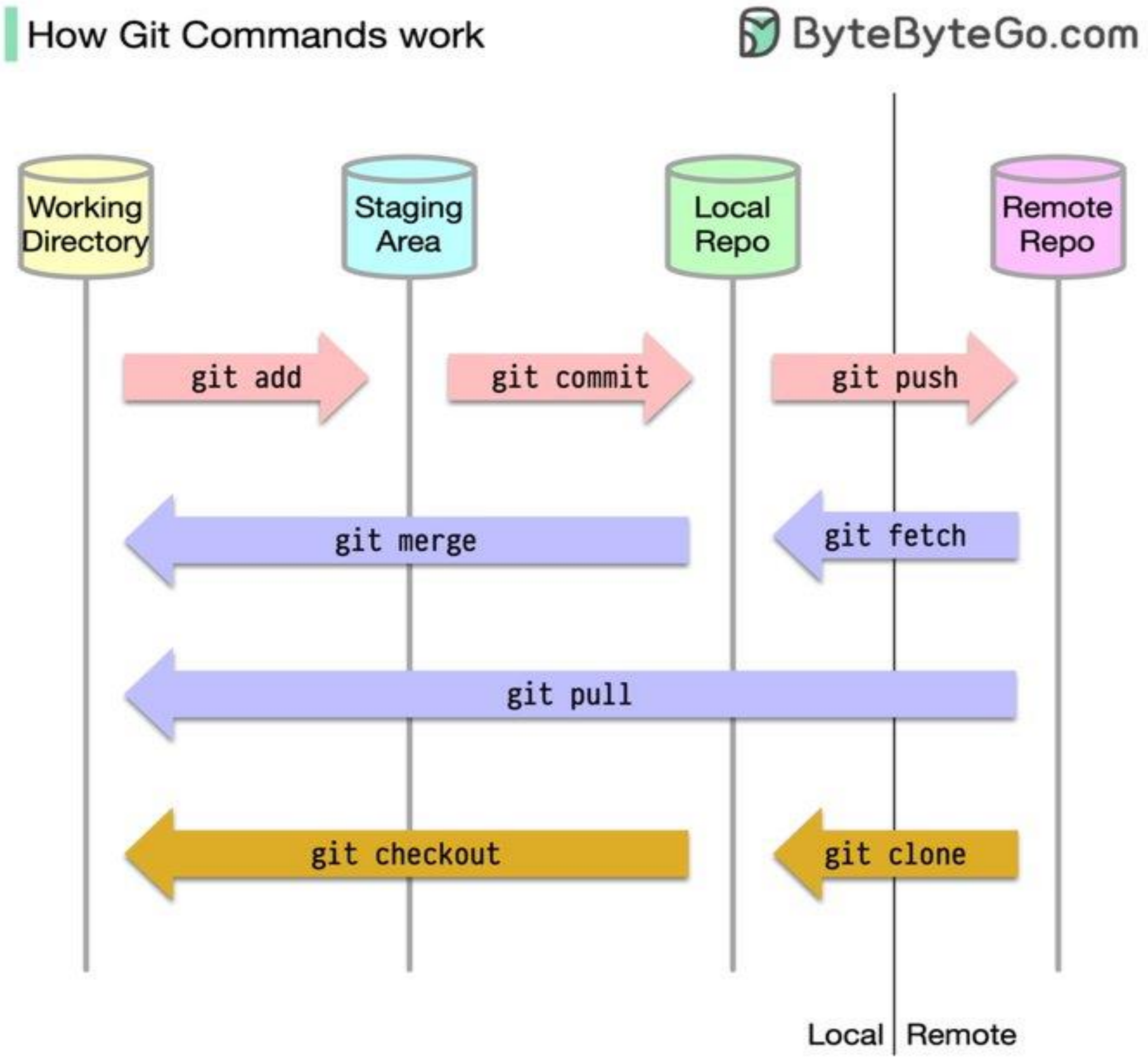
PUSHING TO REMOTE REPO

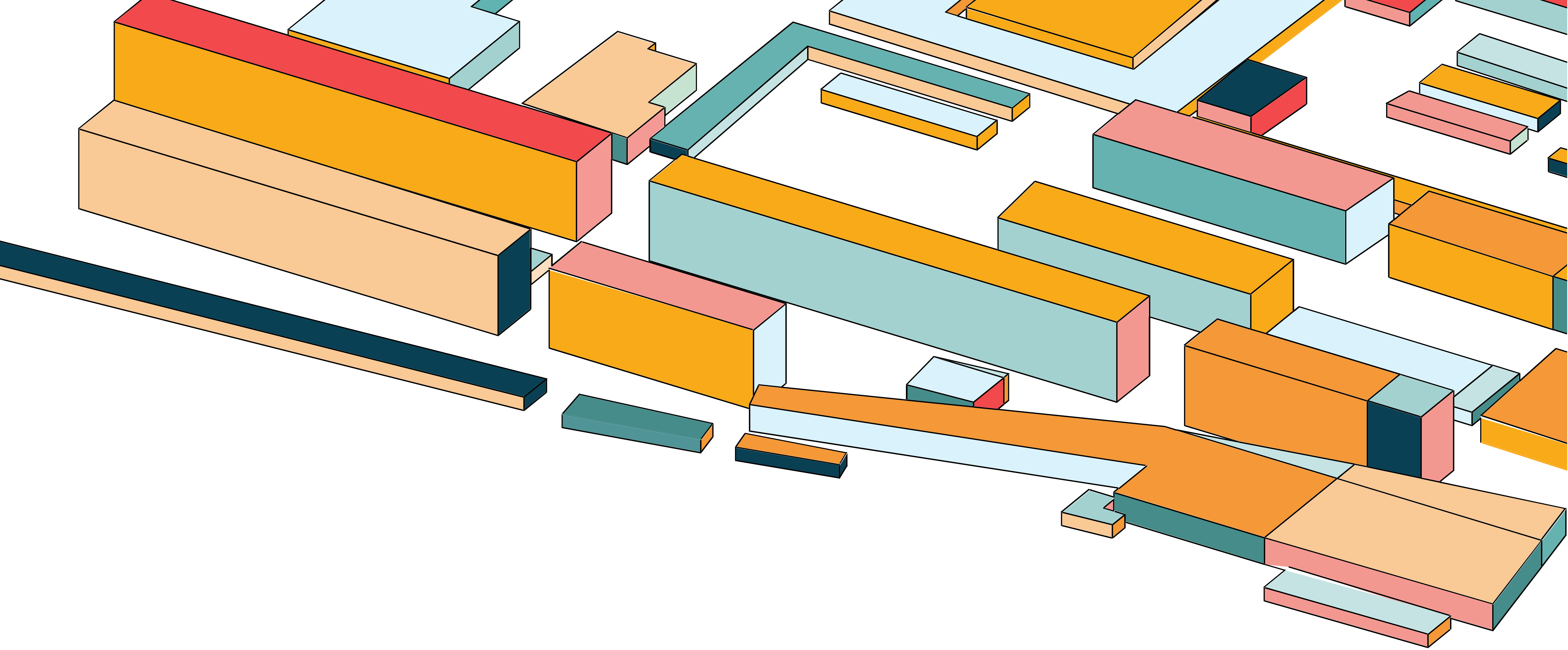
- When you have your project at a point that you want to share, you have to push it upstream
- Command: `git push <remote> <local>`, where `origin` is the default remote name and `main` is the default local branch name
- Push works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime.
- Otherwise, you must fetch their work first and incorporate it into yours before you'll be allowed to push.



PULLING FROM REMOTE REPO

- `git fetch <remote>` fetches any new work that has been pushed to that server since you cloned (or last fetched from) it
- `git fetch` only downloads the data to your local repository, it doesn't automatically merge it with any of your work or modify what you're currently working on. You have to *merge* it manually into your work when you're ready.
- `git pull <remote>` command automatically fetches and then merges that remote branch into your current branch.





GIT INTERNALS

GIT INTERNALS

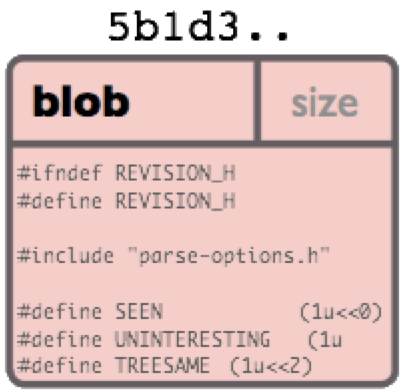
- At the core of Git is a simple key-value data store.
  - Key: hash
  - Value: a sequence of bytes representing files, directories, commits, etc.
  - We can provide a value to git and git will calculate a unique key for it, which can be used later to retrieve the content.
- This key-value structure is persistent, i.e. it’s stored in our disk, the `.git` directory

GIT INTERNALS

- To understand the core of Git internals, there are 3 things to we should know
  - **Objects:** content of files, directories, commits, and tags, identified by SHA-1 hash, stored in `.git/objects/`
  - **References**: A branch, remote branch or a tag, which is simply a pointer to an object, stored in plain text in `.git/refs/`
  - **Index**: a staging area, stored as a binary file in `.git/index.`

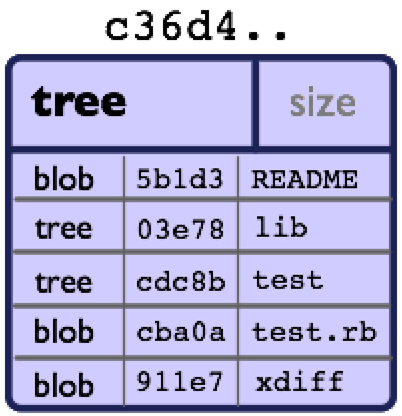
TYPES OF GIT OBJECTS

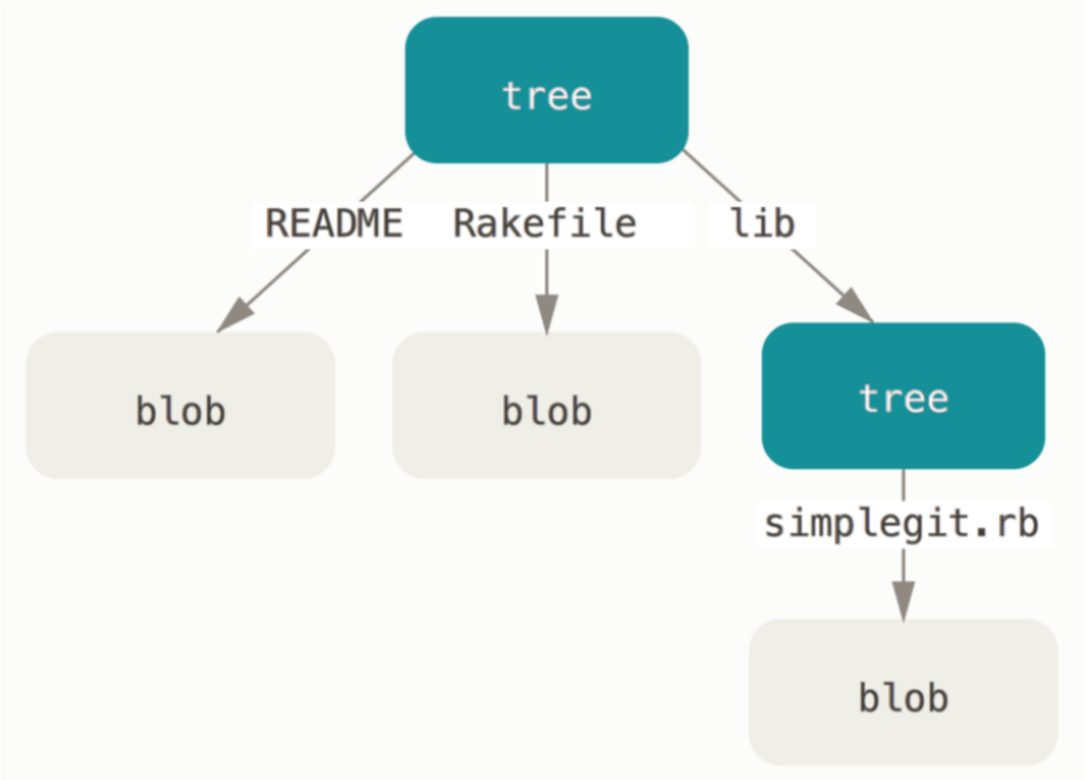
- Blob: file content, identified by a hash
- Tree object
- Commit object
- Tag object



TYPES OF GIT OBJECTS

- Blob: file content, identified by a hash
- **Tree object**: list of pointers to blob (file), or tree (directory), identified by a hash
- Commit object
- Tag object





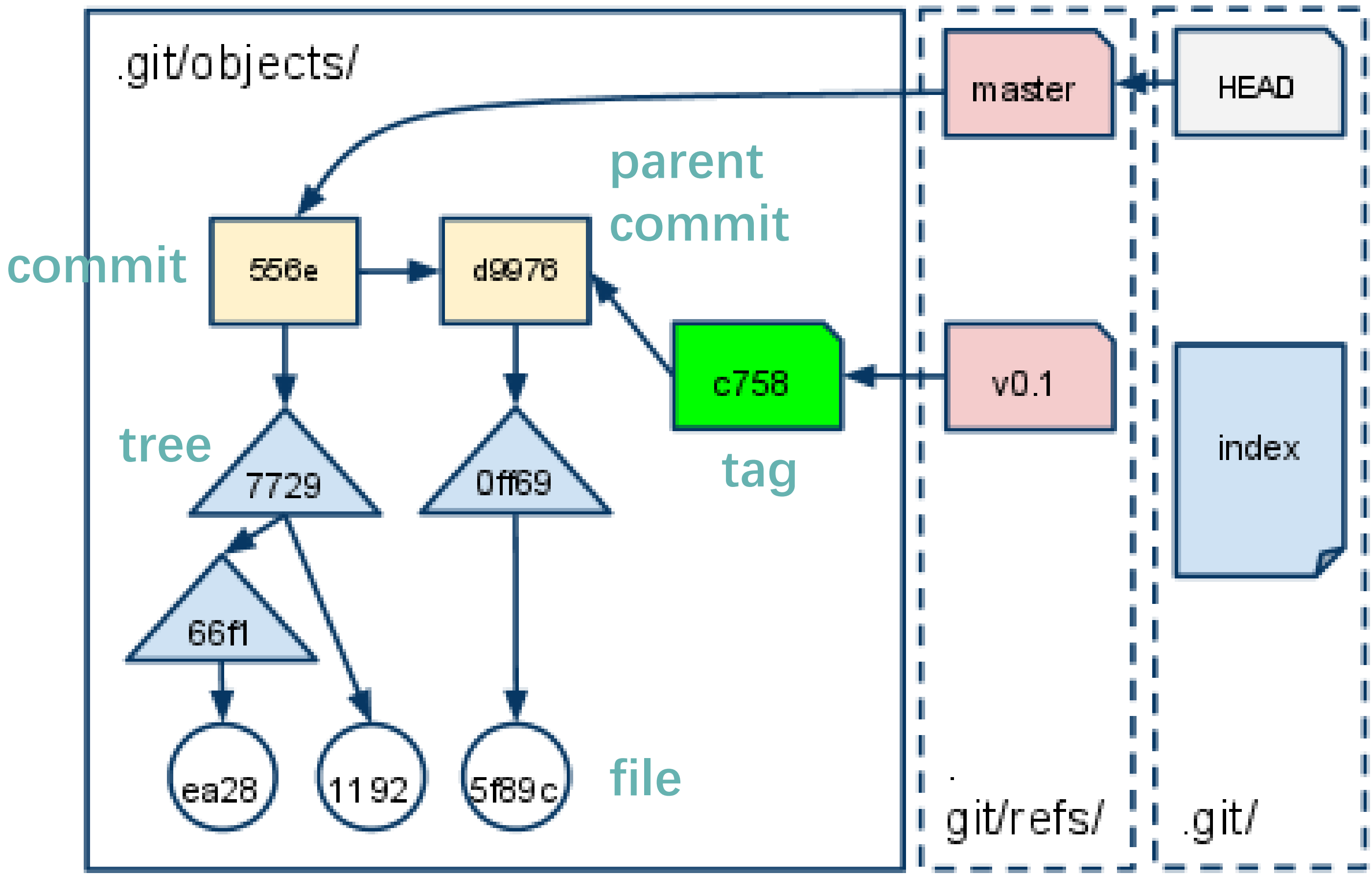
TYPES OF GIT OBJECTS

- Blob: file content, identified by a hash
- Tree object: list of pointers to blob (file), or tree (directory), identified by a hash
- **Commit object**:
  - Reference to the top-level tree for the snapshot of the project at that point
  - Parent commits if any
  - Author info, commit message, etc.
- Tag object

TYPES OF GIT OBJECTS

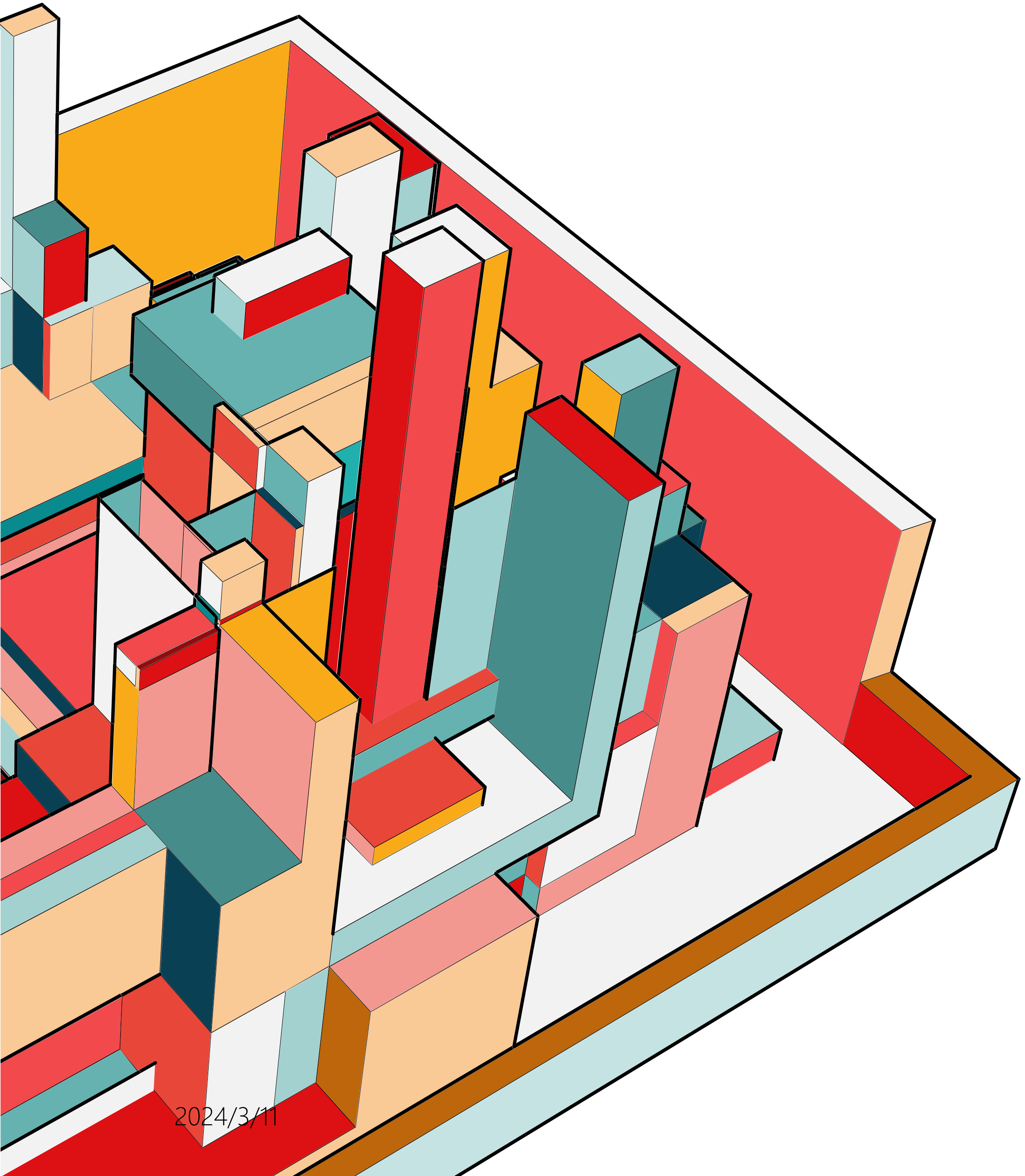
- Blob: file content, identified by a hash
- Tree object: list of pointers to blob (file), or tree (directory), identified by a hash
- Commit object:
  - Reference to the top-level tree for the snapshot of the project at that point
  - Parent commits if any
  - Author info, commit message, etc.
- **Tag object**: name associated with a commit (+ potential metadata)

EXAMPLE



HEAD is a symbolic reference that points to the current branch you are working on.

https://devenderprakash.wordpress.com/2017/01/22/git-internals/

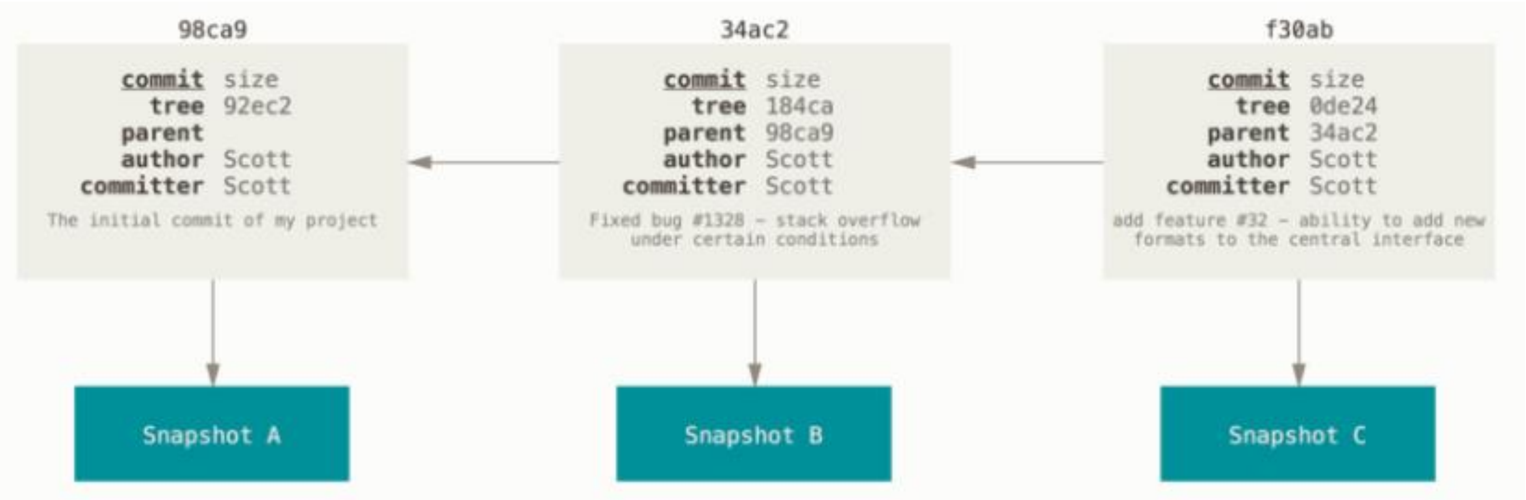


GIT BRANCHING

- “Killer feature” of Git
- Branching means you diverge from the main line of development and continue to do work without messing with that main line
- Git branching is incredibly lightweight: making branches and switching back and forth between branches are nearly instantaneous

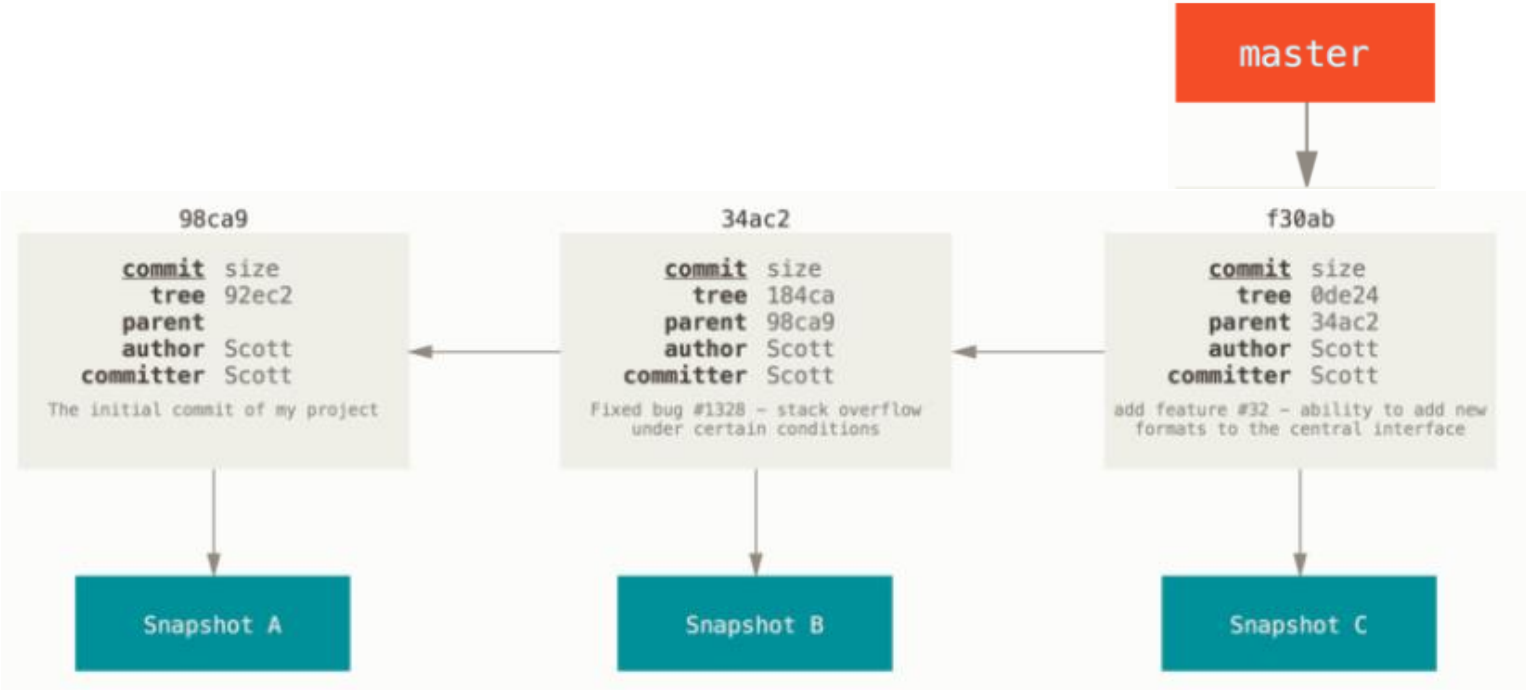
GIT BRANCHING

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



GIT BRANCHING

- A branch in Git is simply a lightweight movable **pointer** to one of these commits.
- The default branch name in Git is `master/main`. As you start making commits, you're given a master branch that points to the last commit you made.
- Every time you commit, the master branch pointer moves forward automatically.

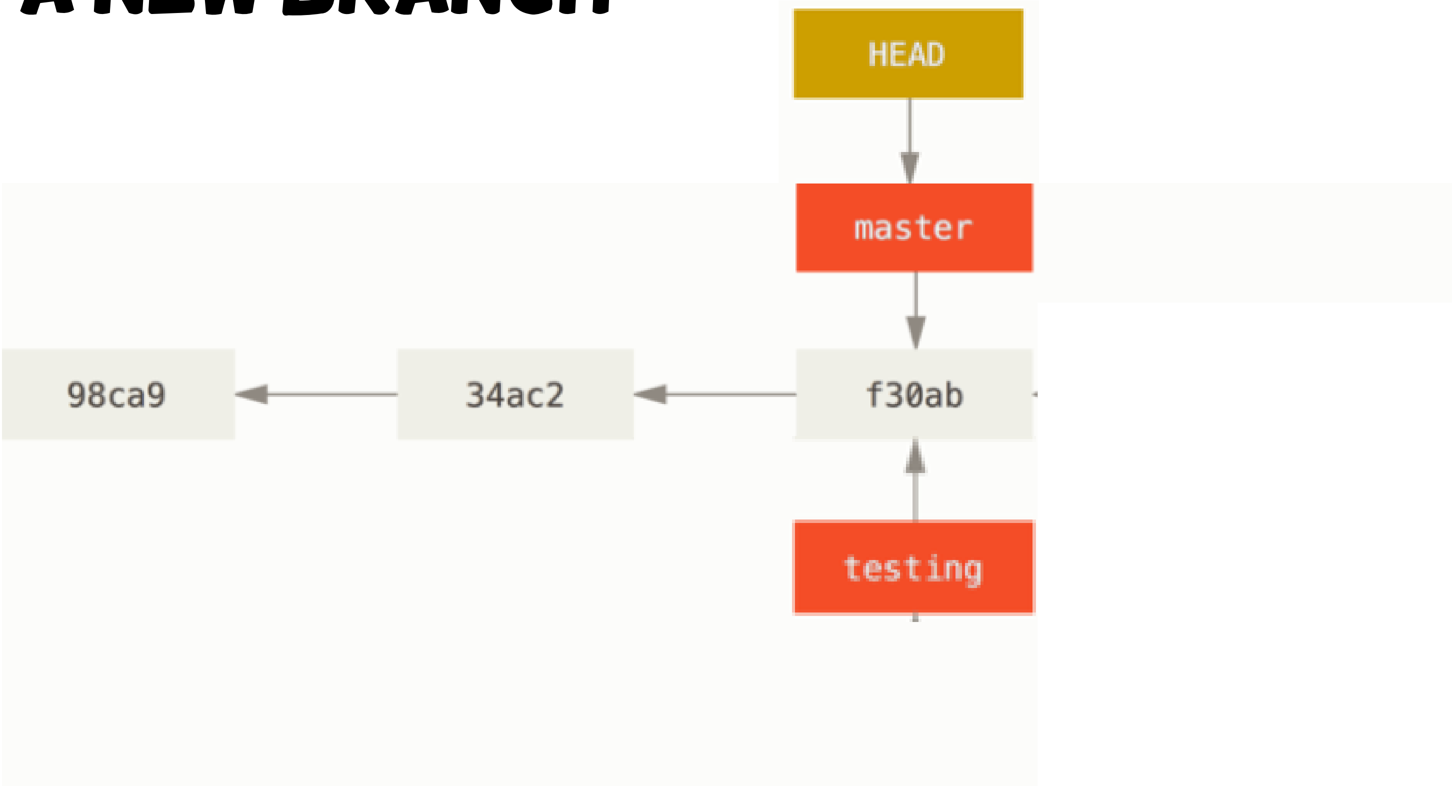


CREATING A NEW BRANCH

- Git uses a special pointer, HEAD, to track which branch you're currently on

- Use `git branch <branch>` to create a new pointer to the same commit you're currently on

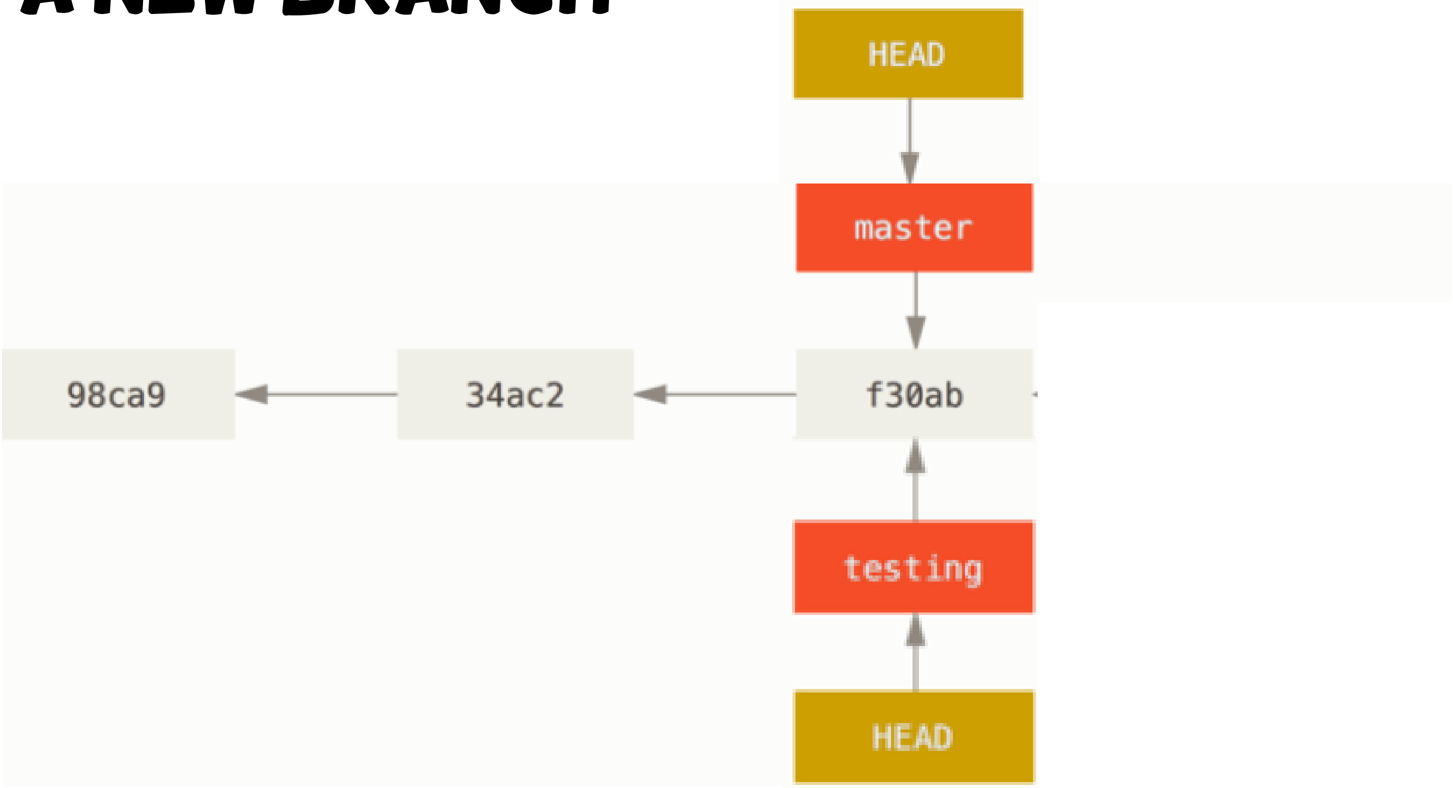
- The `git branch` command only created a new branch — **it didn't switch to that branch.**



https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

CREATING A NEW BRANCH

- Use **`git checkout <branch>`** to switch branch, which moves HEAD to point to <branch>



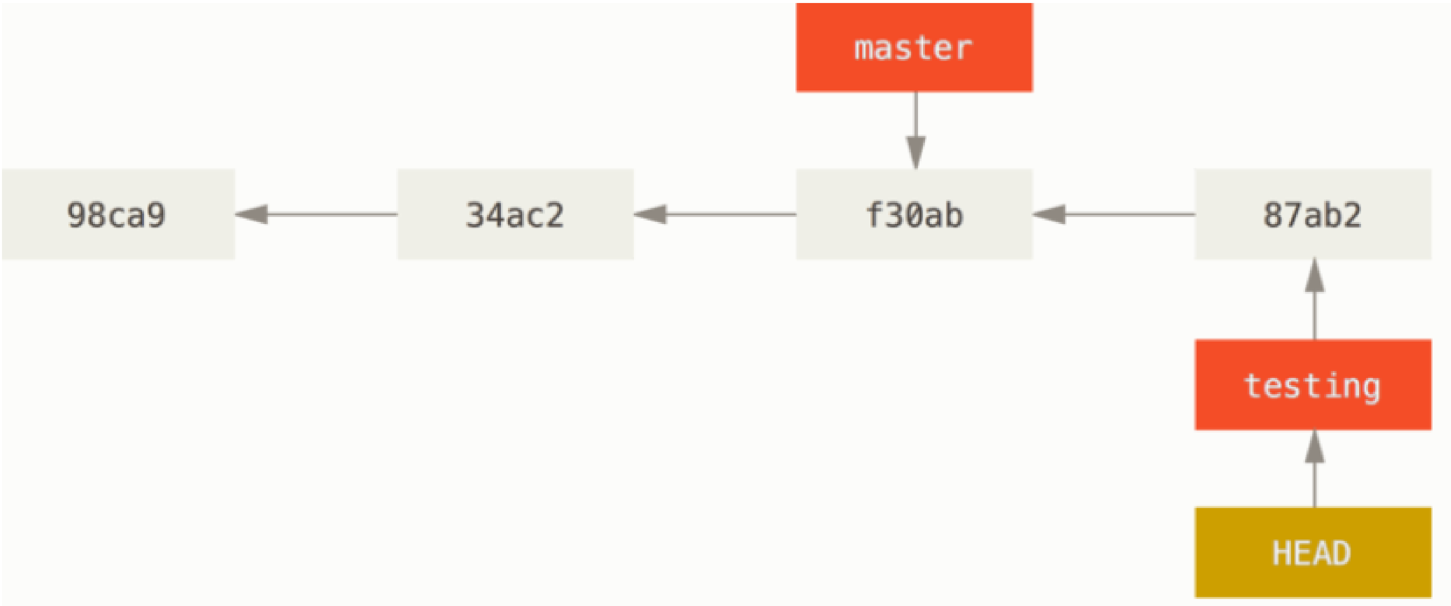
https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

CREATING A NEW BRANCH

- Use **`git checkout <branch>`** to switch branch, which moves HEAD to point to <branch>

- Now, making a new commit will make the **`testing`** branch move forward, while the master branch stays the same

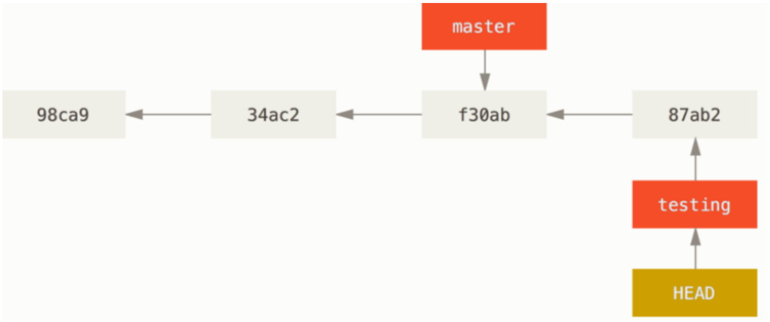
- Use **`git push origin <branch>`** to push the new branch to remote



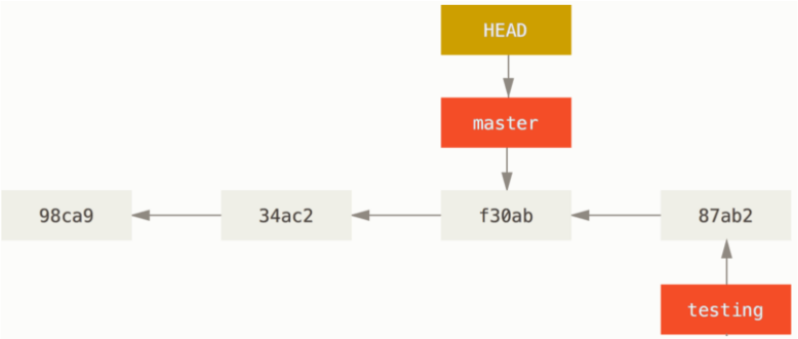
https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

DIVERGED HISTORY

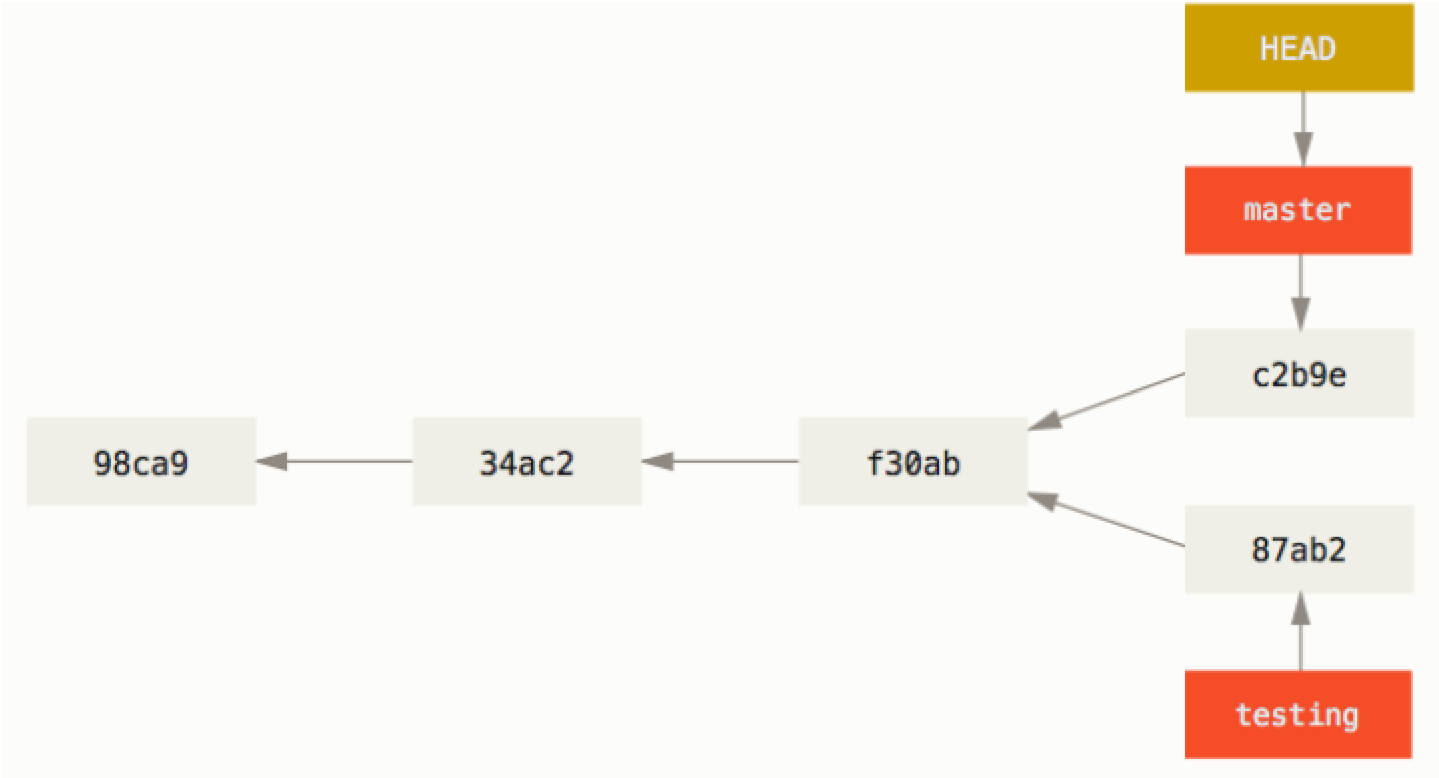
1. Current status



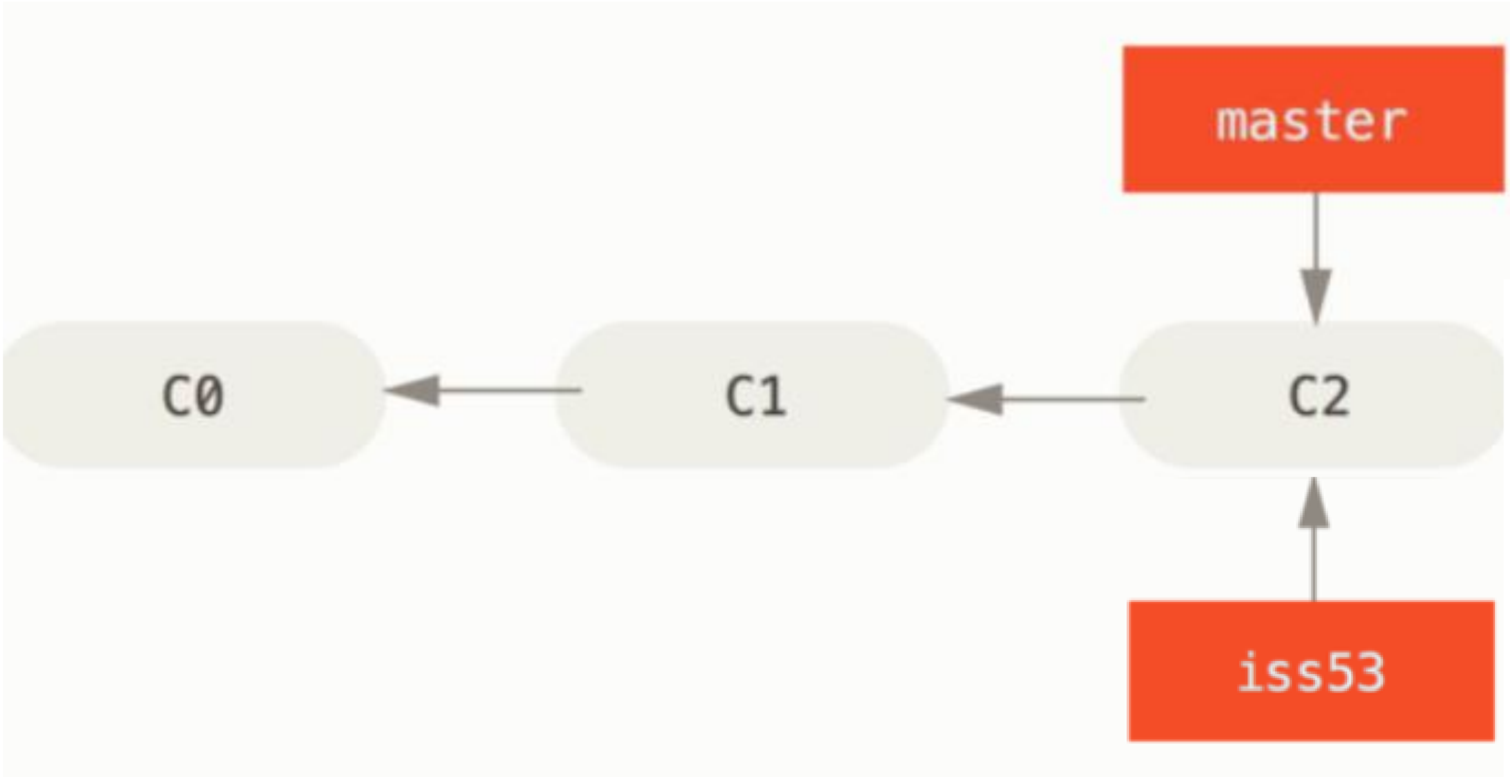
2. Switch back to the “master” branch



3. Make a commit in the “master” branch



MERGING BRANCHES




```
$ git checkout -b iss53
Switched to a new branch "iss53"
```

This is shorthand for:

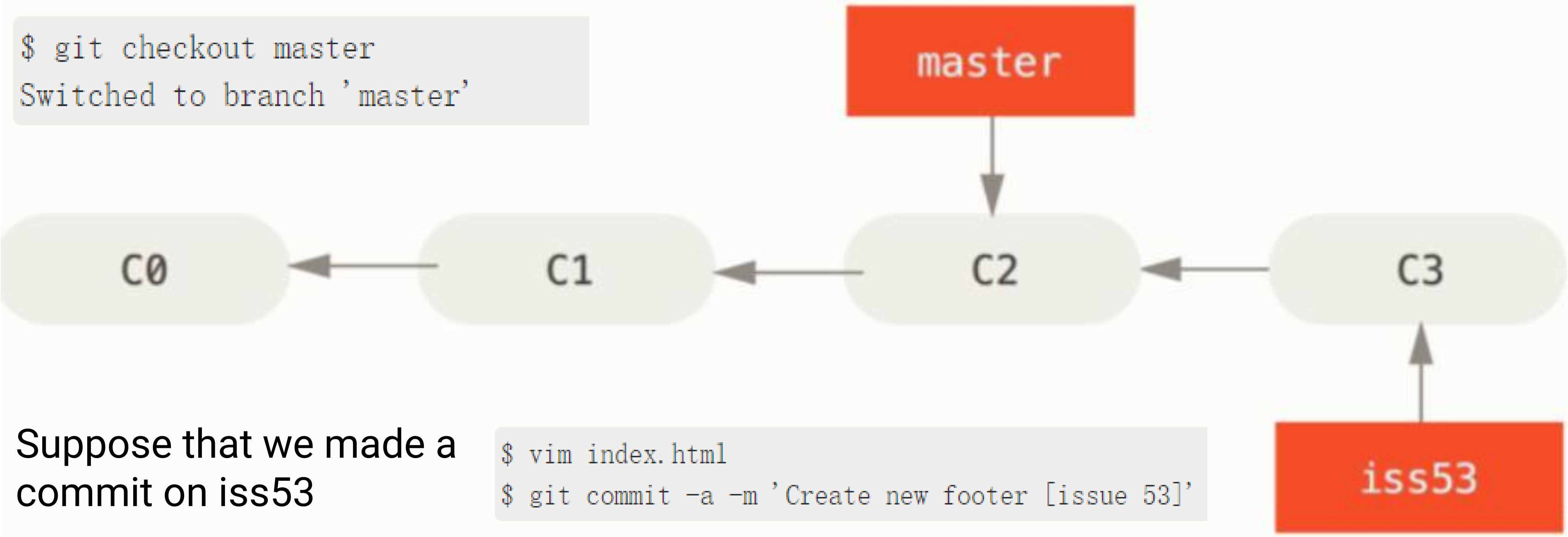
```
$ git branch iss53
$ git checkout iss53
```

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

MERGING BRANCHES

We can also switch back to master

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

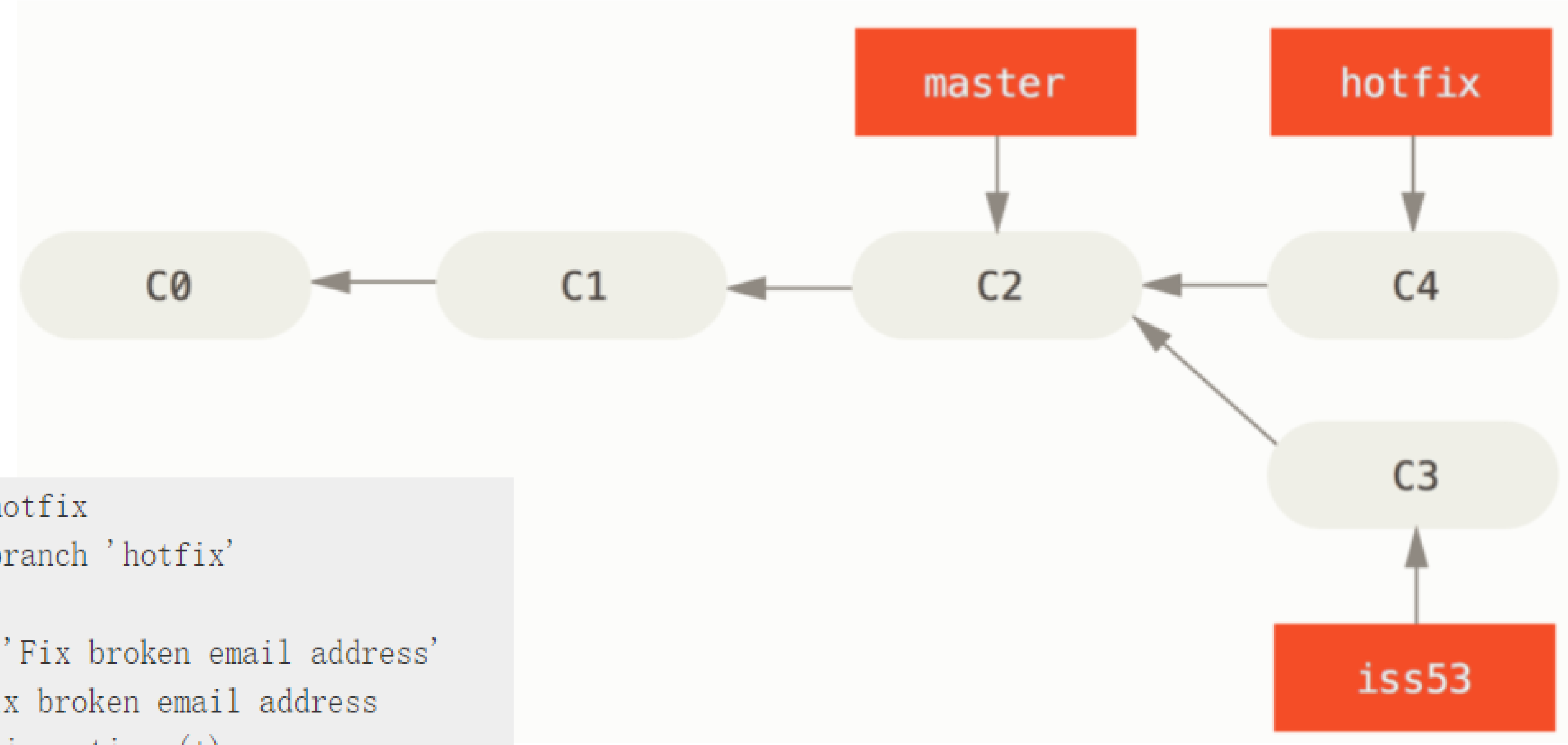


Suppose that we made a commit on iss53

```
$ vim index.html
$ git commit -a -m 'Create new footer [issue 53]'
```

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

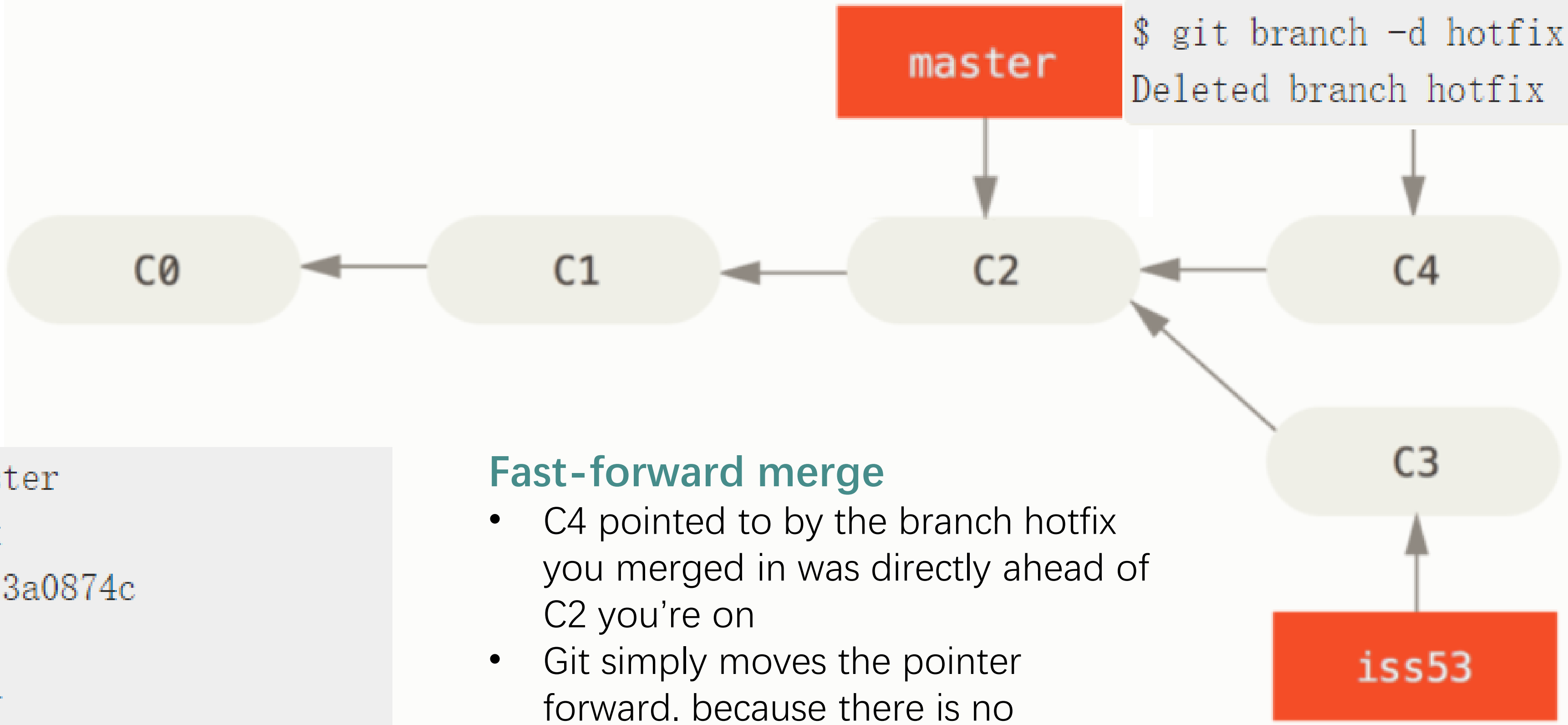
MERGING BRANCHES




```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'
[hotfix 1fb7853] Fix broken email address
 1 file changed, 2 insertions(+)
```

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

MERGING BRANCHES




```
$ git branch -d hotfix
Deleted branch hotfix (3a0874c).
```

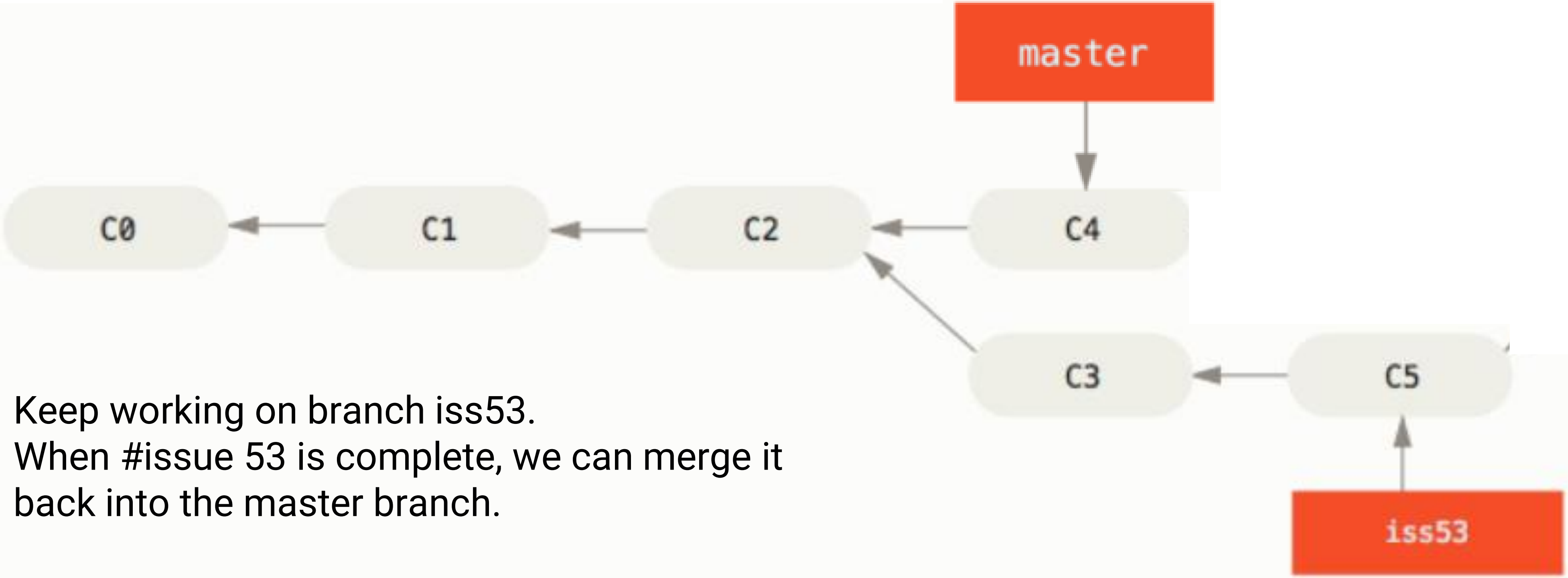
```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

Fast-forward merge

- C4 pointed to by the branch hotfix you merged in was directly ahead of C2 you're on
- Git simply moves the pointer forward. because there is no divergent work to merge together

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

MERGING BRANCHES

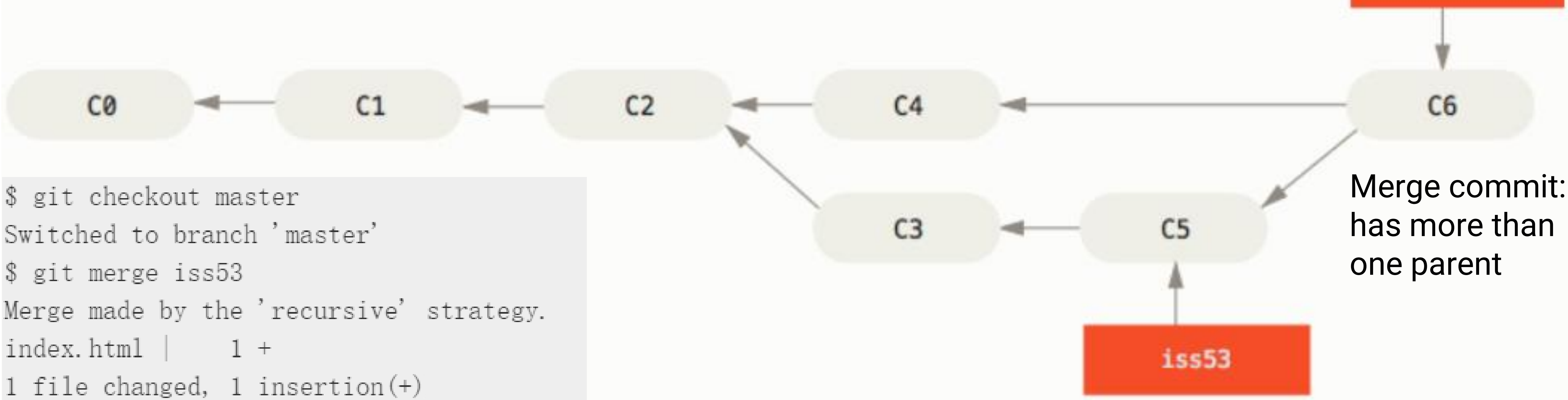


Keep working on branch iss53.
When #issue 53 is complete, we can merge it back into the master branch.

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

MERGING BRANCHES

3-way merge: Git does a simple 3-way merge, using the two snapshots pointed to by the branch tips and the most recent common ancestor of the two.



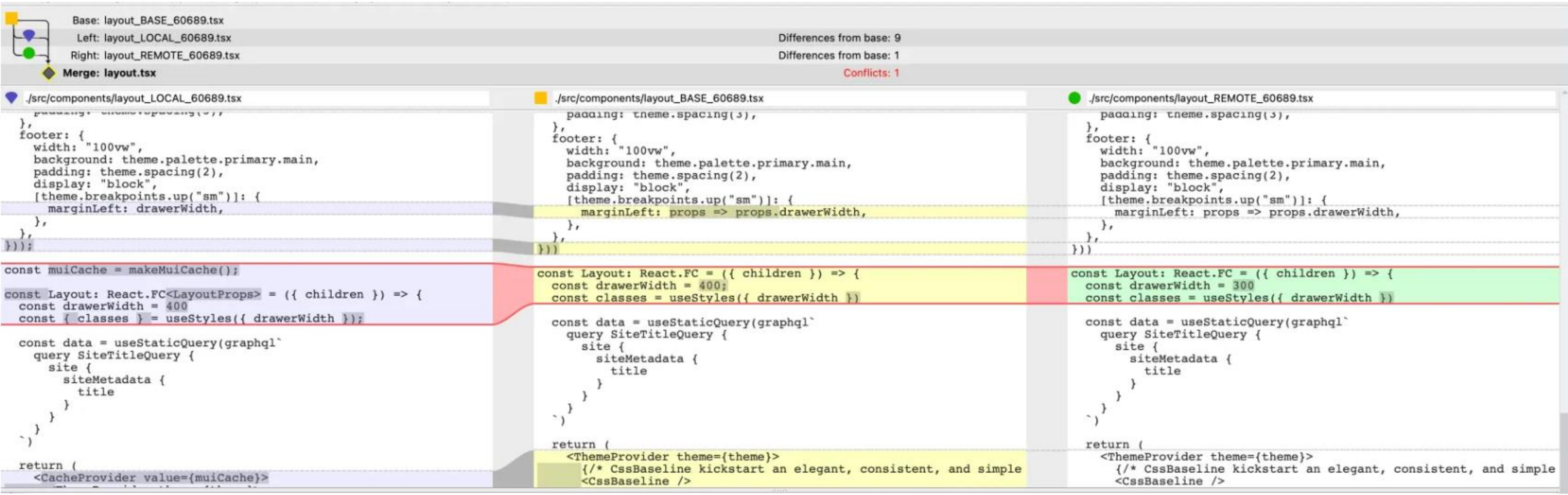

```
$ git checkout master
Switched to branch 'master'
$ git merge iss53
Merge made by the 'recursive' strategy.
index.html |    1 +
1 file changed, 1 insertion(+)
```

Merge commit: has more than one parent

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

MERGE CONFLICTS

When there are conflicting changes, git cannot do merge automatically and you have to resolve the conflict manually.



TOO COMPLICATED?

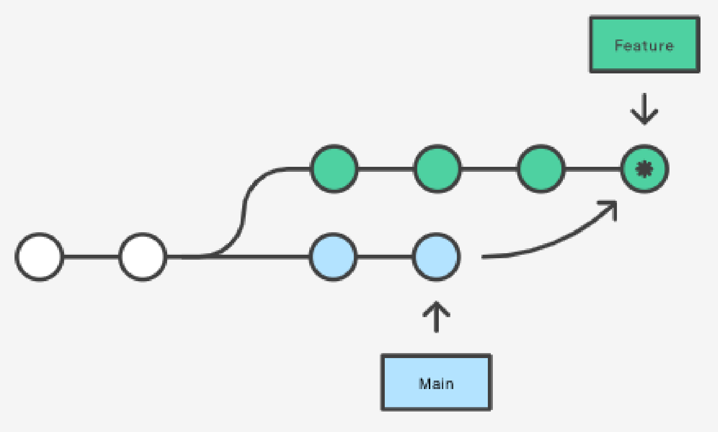
https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

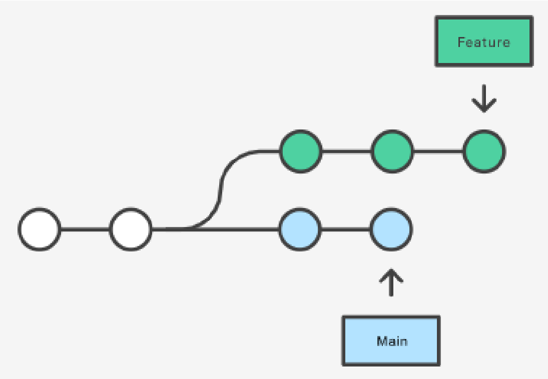
MERGING VS. REBASING

- In addition to `git merge`, you can also use `git rebase` to integrate changes from one branch into another
- With rebase, you can take all changes committed on one branch and replay them on a different branch.

Merge the main branch into the feature branch

```
git checkout feature
git merge main
```

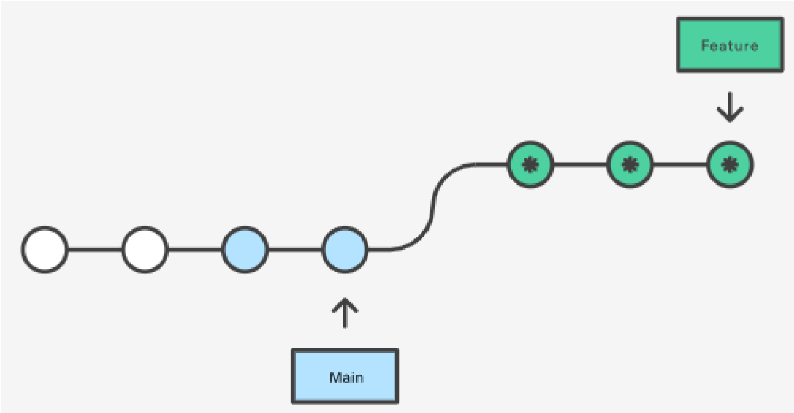




https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Rebase the feature branch onto the main branch

```
git checkout feature
git rebase main
```

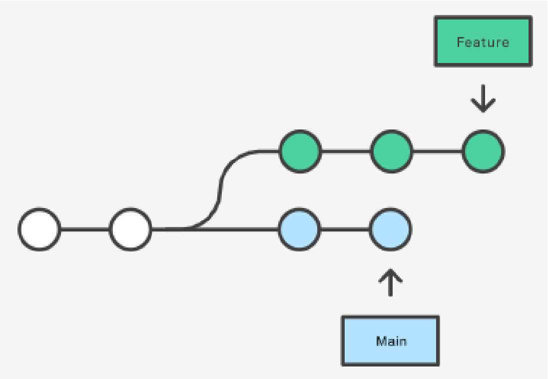


MERGING VS. REBASING

- In addition to `git merge`, you can also use `git rebase` to integrate changes from one branch into another
- With rebase, you can take all changes committed on one branch and replay them on a different branch.

- Rebase moves the entire feature branch to begin on the tip of the main branch, effectively **incorporating all the new commits in main**.

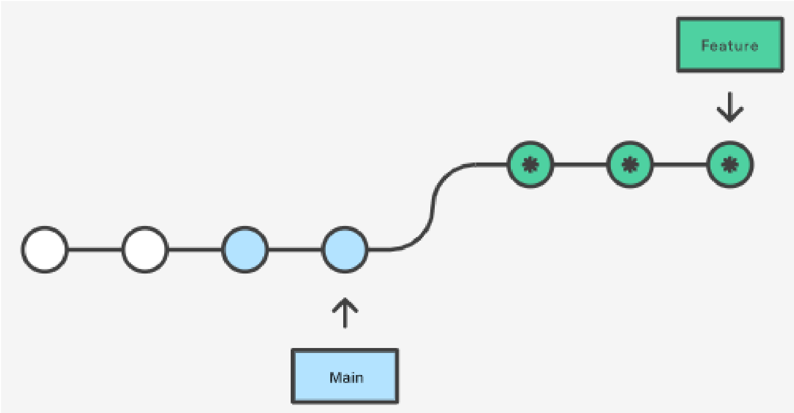
- But, instead of using a merge commit, rebasing **re-writes the project history** by creating brand new commits for each commit in the original branch.

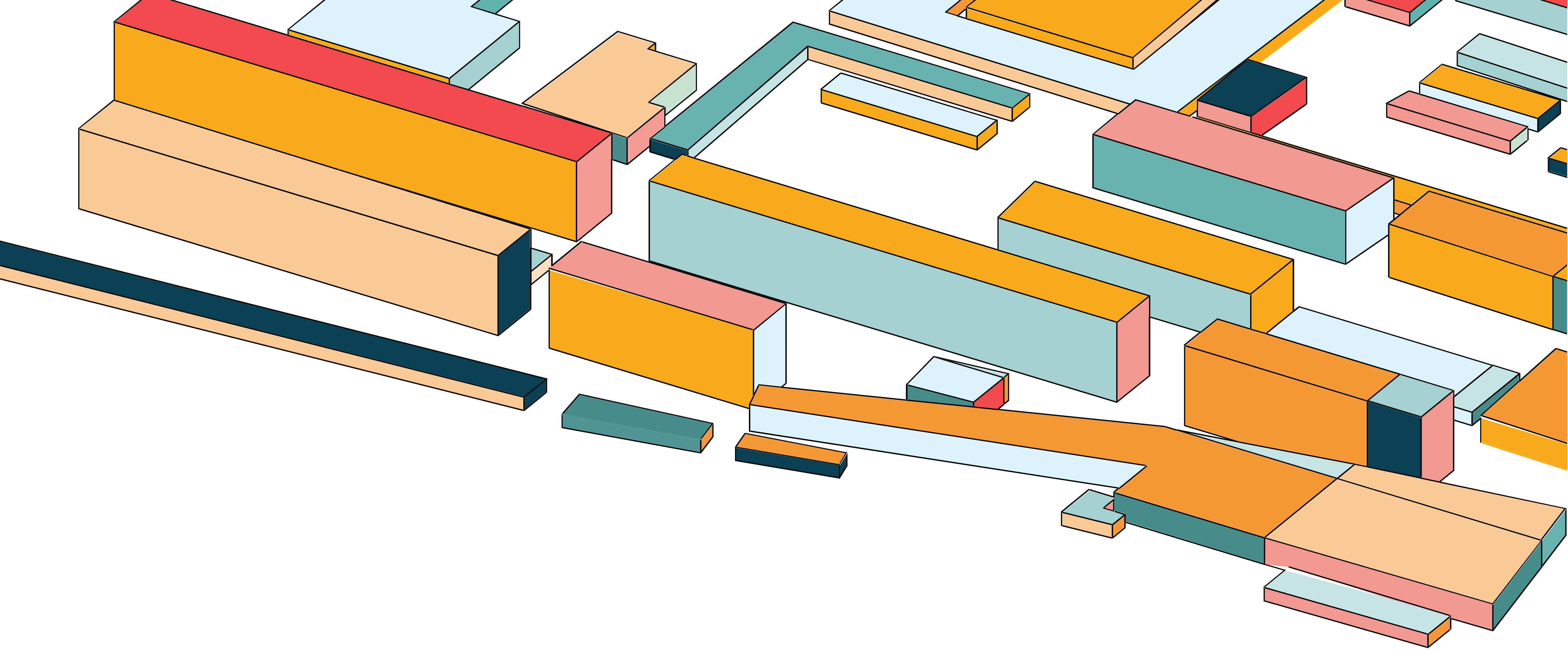


https://www.atlassian.com/git/tutorials/merging-vs-rebasing

Rebase the feature branch onto the main branch

```
git checkout feature
git rebase main
```

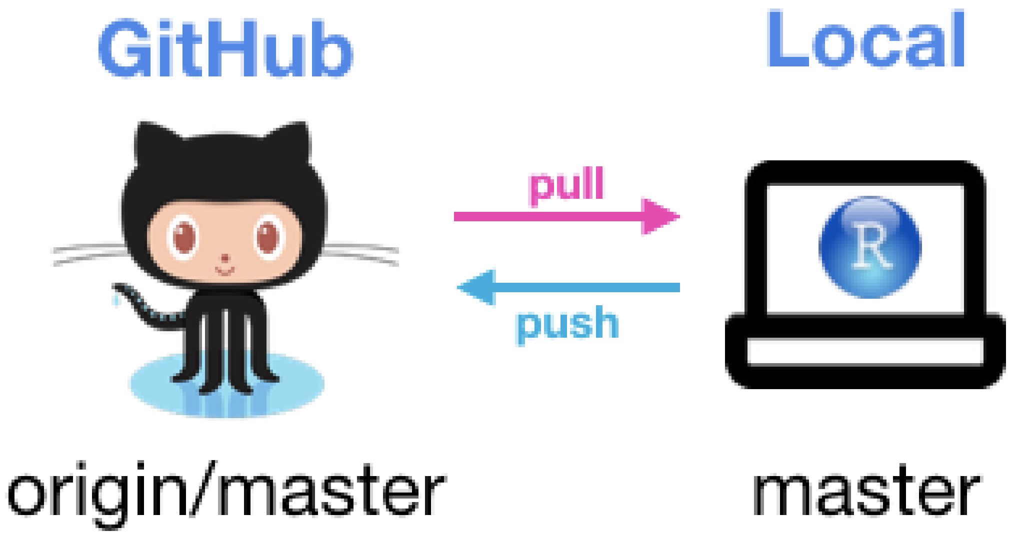




GITHUB WORKFLOW

GITHUB WORKFLOW I

No-branch workflow

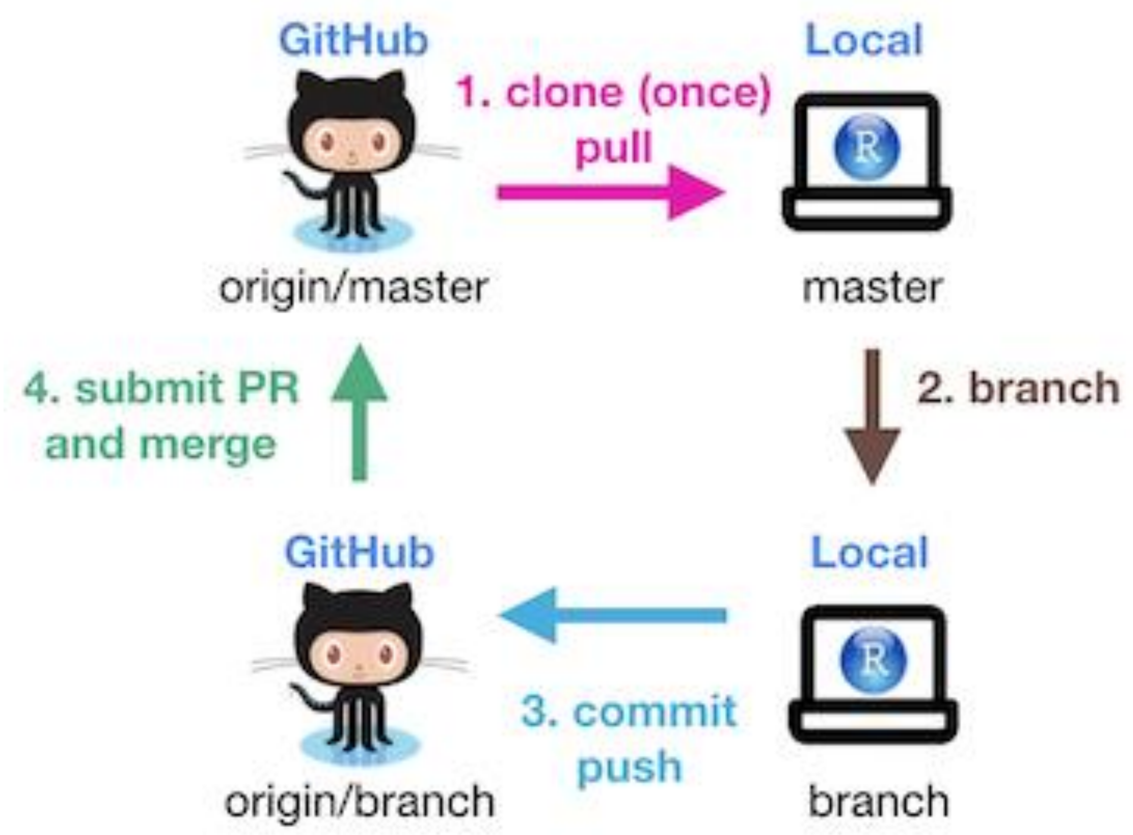


https://jtr13.github.io/EDAV/github.html

GITHUB WORKFLOW II

Your own repo (or you have write access) with branching.

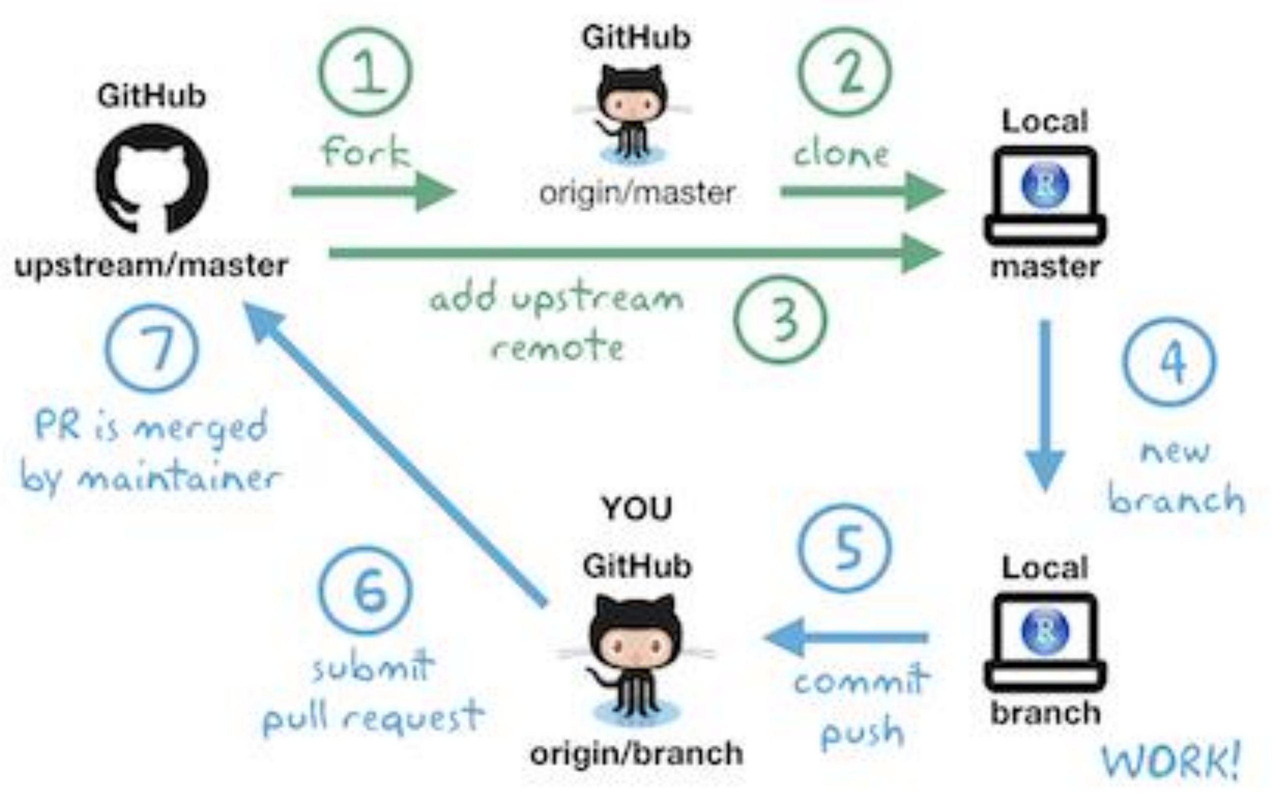
A pull request (PR) is a proposal to merge a set of changes from one branch into another.



https://jtr13.github.io/EDAV/github.html

GITHUB WORKFLOW III

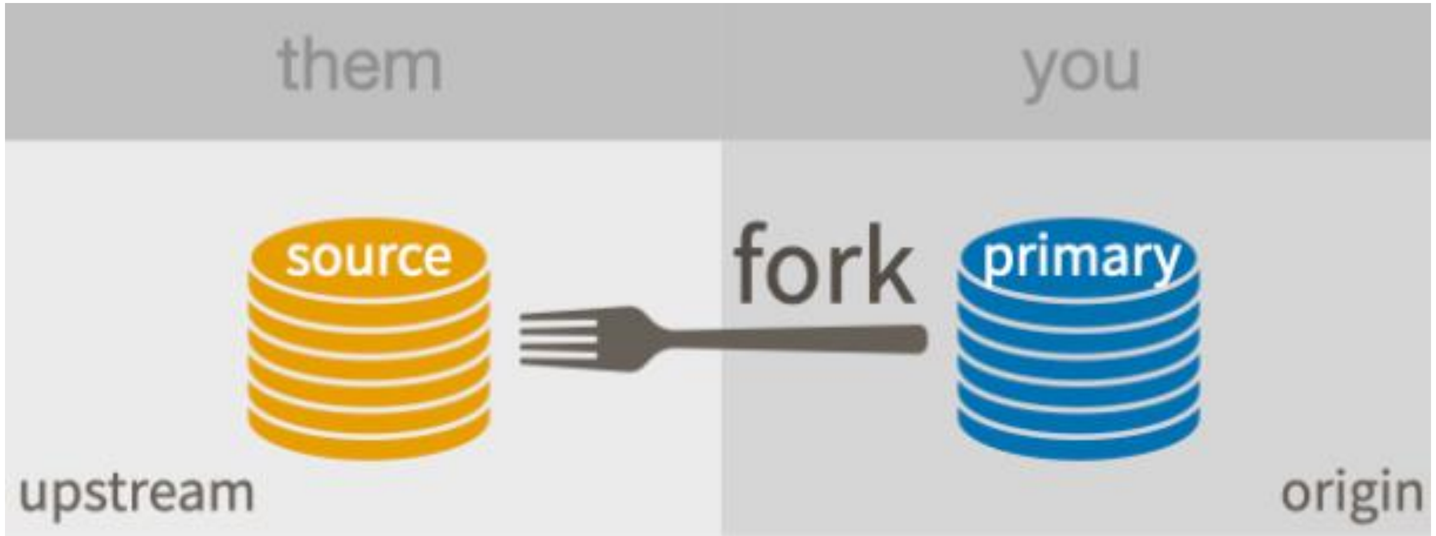
You want to **contribute** to others' repo that you don't have write access to.



https://jtr13.github.io/EDAV/github.html

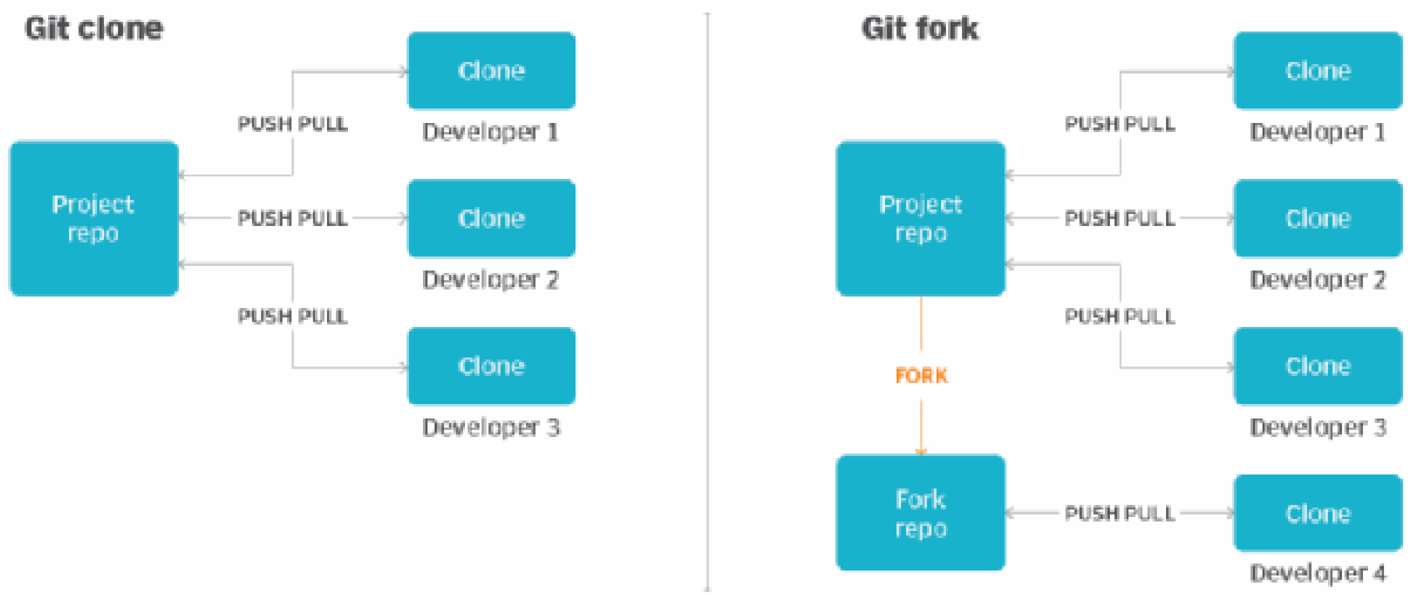
FORK

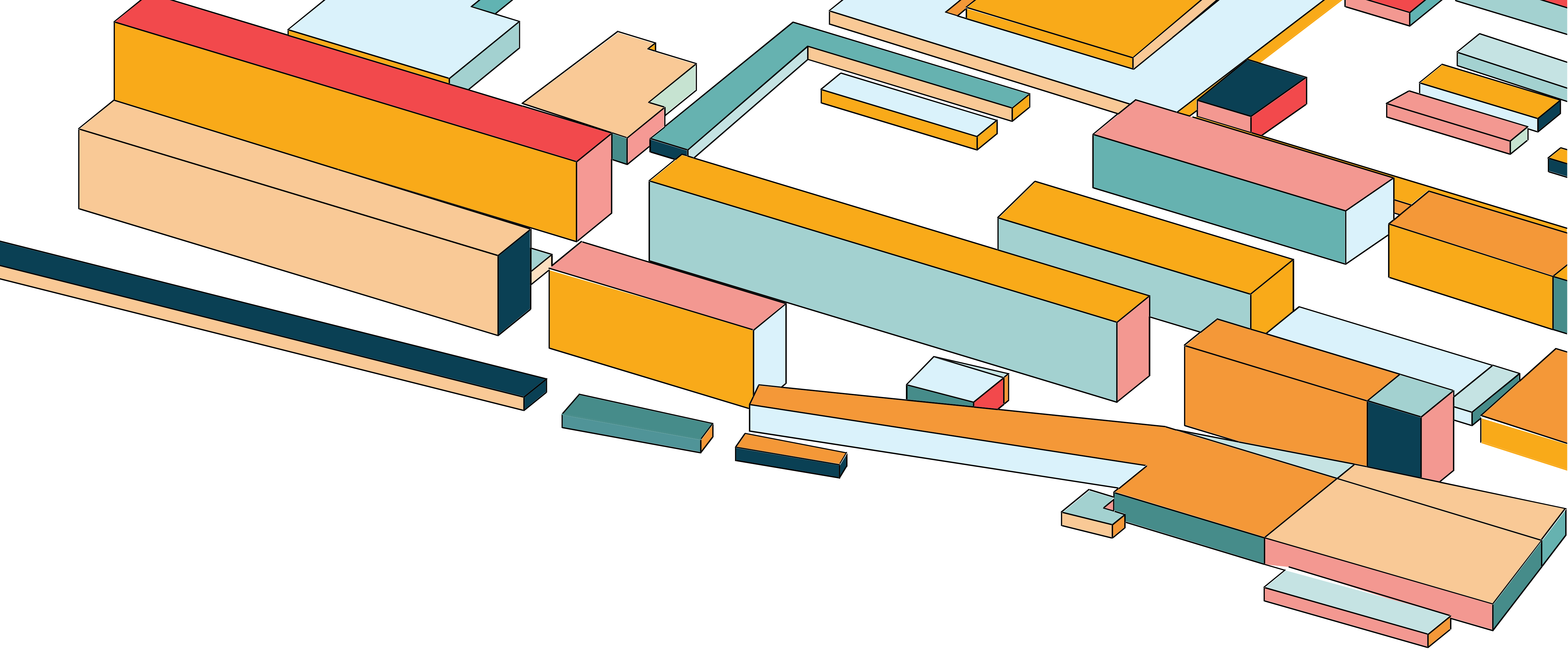
- A Git fork operation will create a completely new copy of the target repository (upstream repo).
- The developer who performs the fork will have complete control over the newly copied codebase.
- Developers who contributed to the upstream repo cannot contribute to the new fork unless access granted



GIT CLONE VS. FORK

- Any public git repository can be forked or cloned
- Key difference: how much control and independence you want over the codebase once you've copied it.
  - Clone: a clone creates a linked copy that will continue to synchronize with the target repository.
  - Fork: A fork creates a completely independent copy of Git repo, disconnecting the codebase from previous committers.





GIT BEST PRACTICES

GIT BEST PRACTICES

Don’t commit generated files or dependencies

- Only commit files that take your manual efforts to create (e.g., `.java`)
- Don’t commit files that can be generated at any time (e.g., `.class`). They normally don’t work with `diff` tools
- Don’t commit dependencies, which will increase the size of your repo. Use tools (e.g., `maven`) to manage dependencies.

GIT BEST PRACTICES

Don’t commit generated files or dependencies

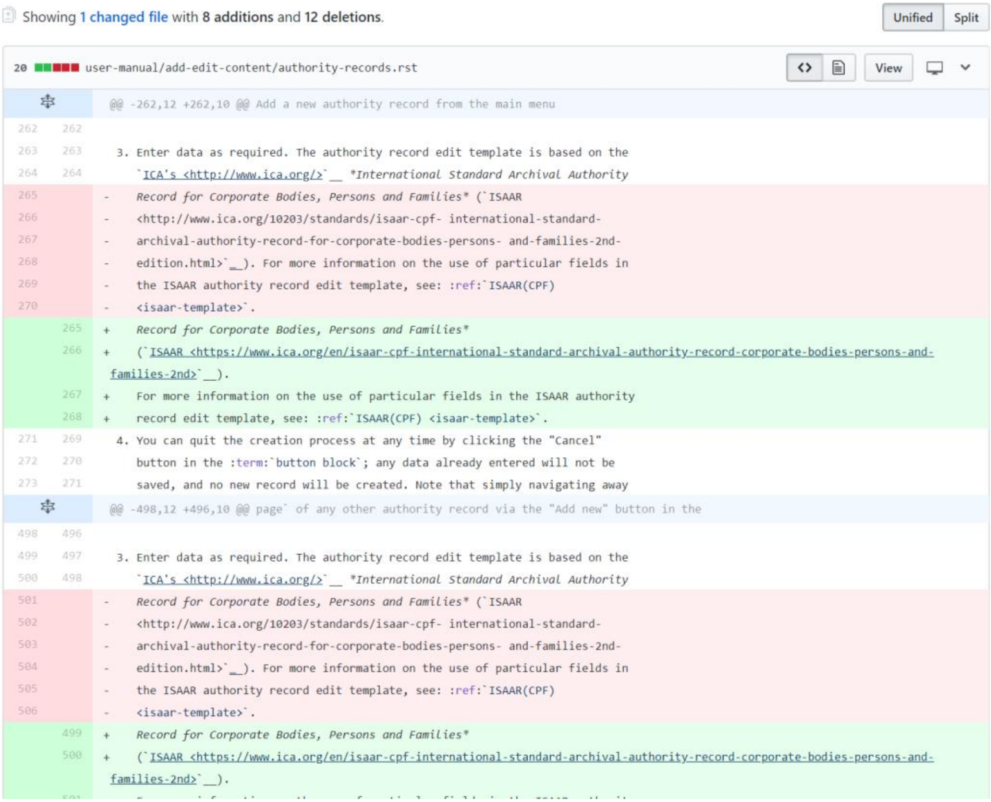
add a `.gitignore` file in your repository's root to automatically tell Git which files or paths you don't want to track.

```
# Compiled class file
*.class

# Log file
*.log

# BlueJ files
*.ctxt

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.nar
*.ear
*.zip
*.tar.gz
*.rar

# virtual machine crash logs, see http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
```

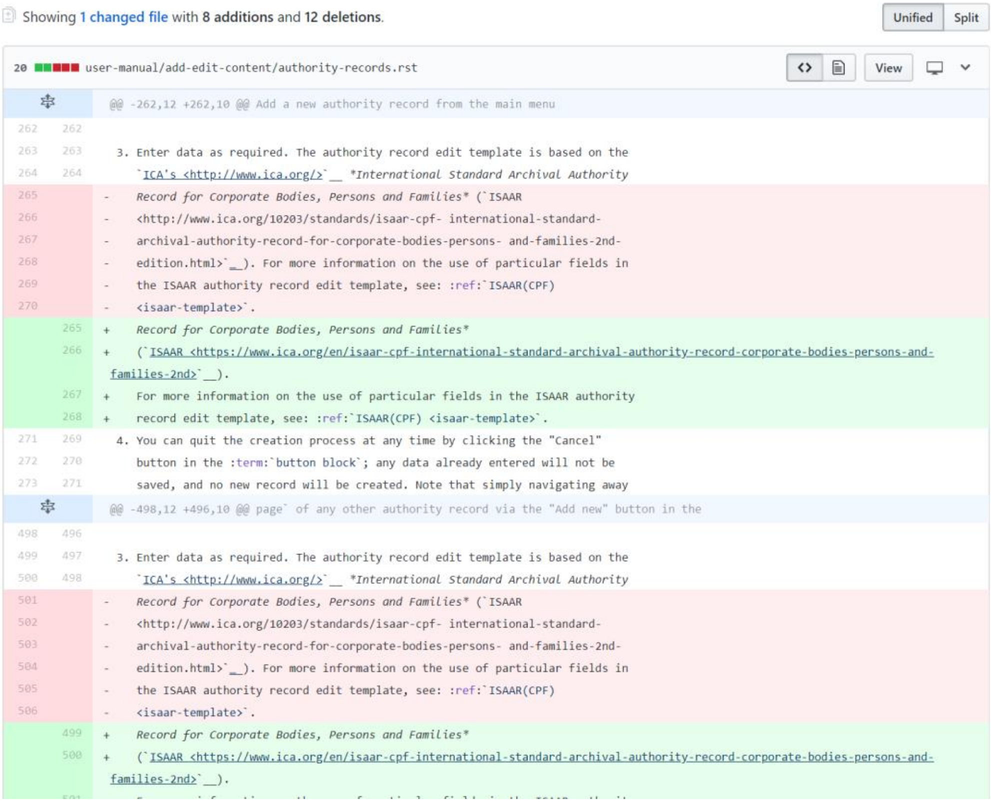
GIT BEST PRACTICES

Make clean, single-purpose commits

- A commit should address a single, atomic issue (e.g., a bug fix), instead of addressing many issues (e.g., a bug fix + a new feature + refactorings + ……)
- Small changes should generally be limited to ~200 LoC



GIT BEST PRACTICES



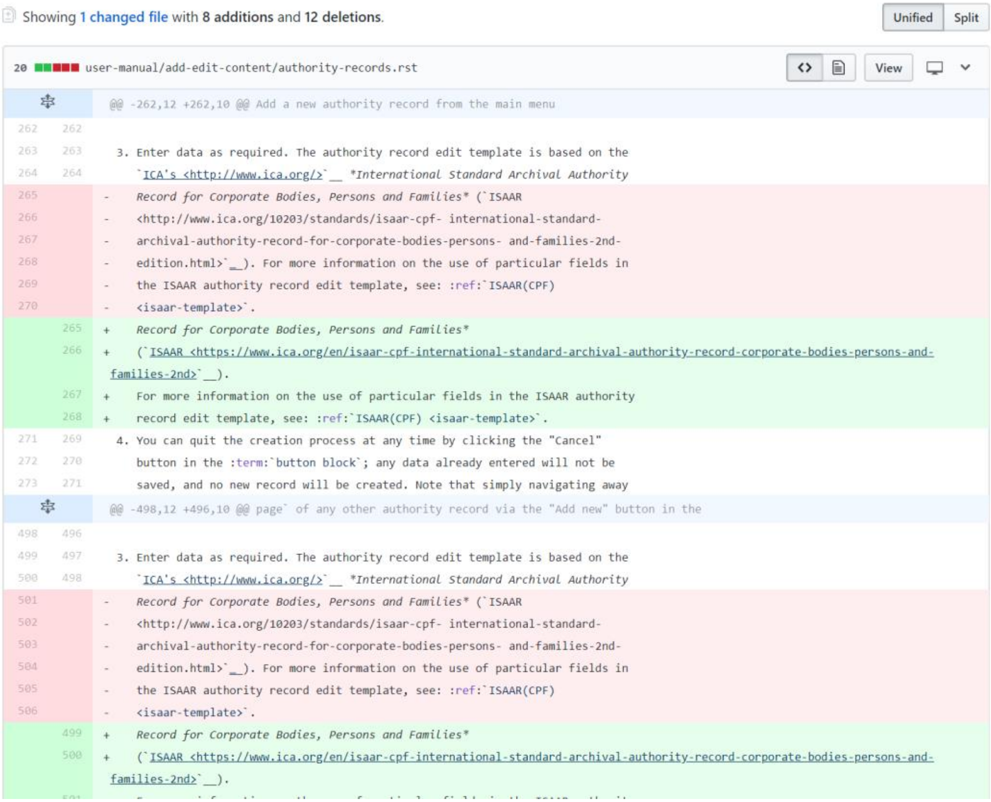
Make clean, single-purpose commits

- Small code changes are easy to digest
  - 35% of the changes at Google are to a single file
- Small code changes allow quick approvals and quicker changes to the codebase
  - Most changes at Google are expected to be reviewed within a day

GIT BEST PRACTICES

Commit early, commit often

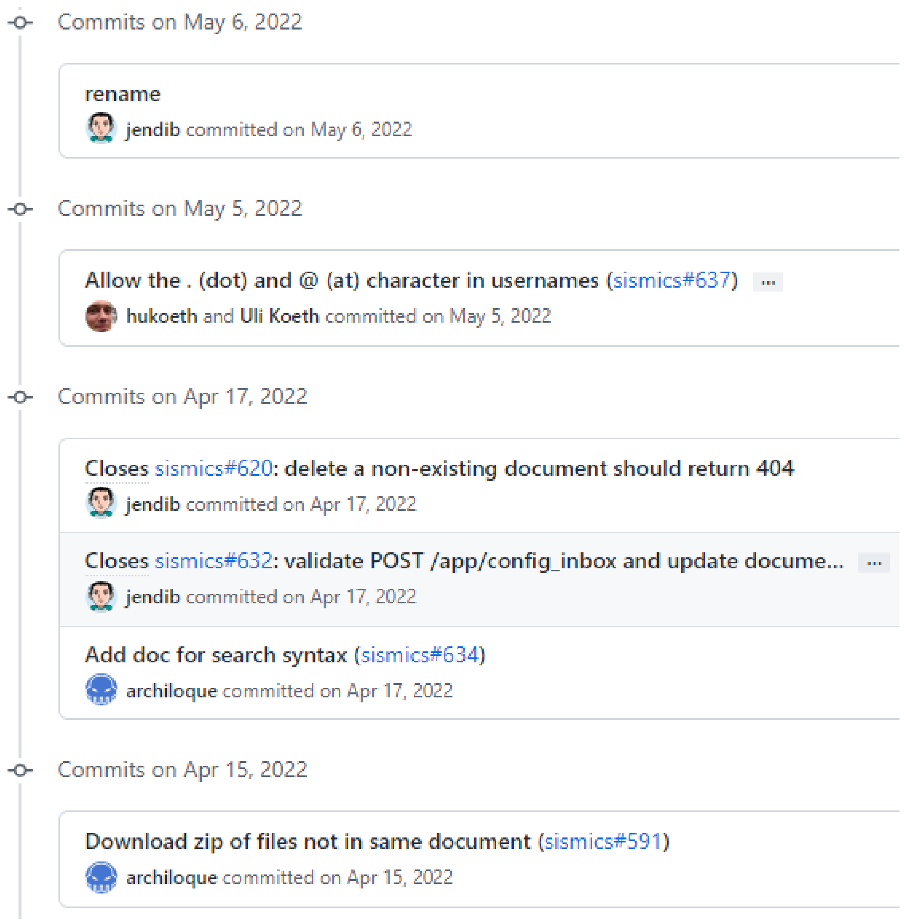
- It is better to work in small chunks and keep committing your work.
- It helps you from losing work, reverting changes, and helping trace what you did
- It helps you keep your code updated with the latest changes so that you avoid conflicts.



GIT BEST PRACTICES

Write meaningful commit messages

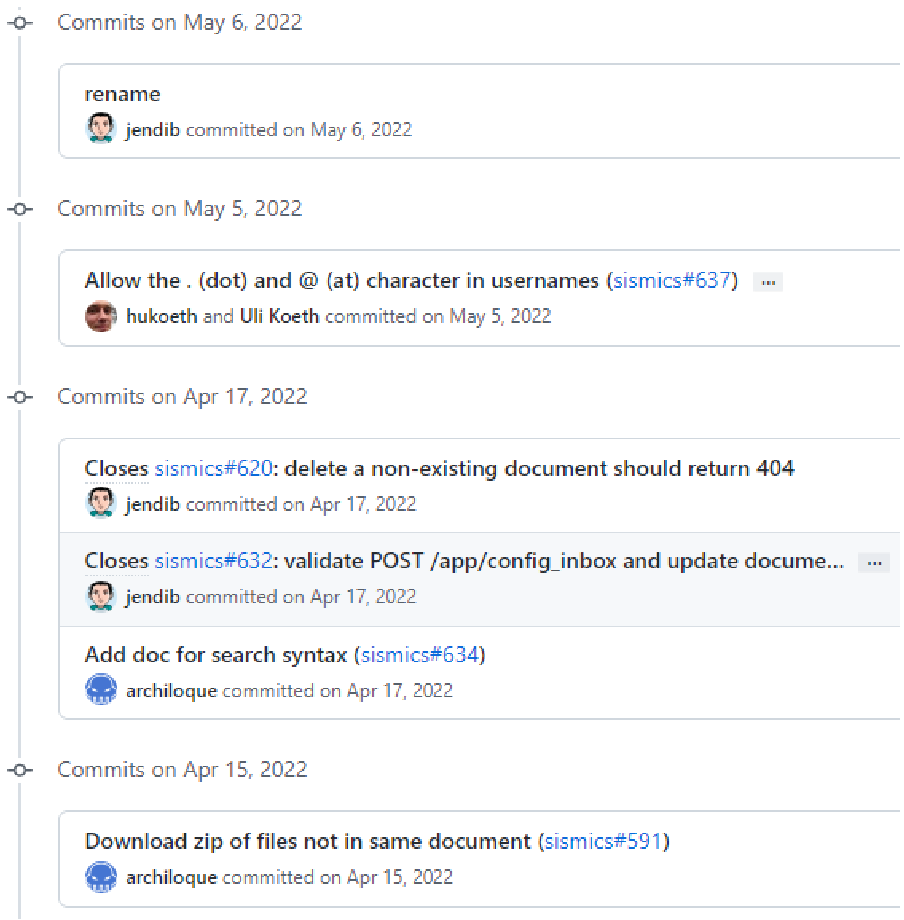
A good commit message should
- summarize the change
- explain what is changed and why
- e.g., “bug fix” is not a good change description



GIT BEST PRACTICES

Write meaningful commit messages

- Insightful and descriptive commit messages make life easier for others as well as your future self.
- A good commit message also allows Code Search tools to locate changes



NEXT

- Software Architecture