Linux Systems and Open Source Software

Version Control System: Git

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Fall 2021

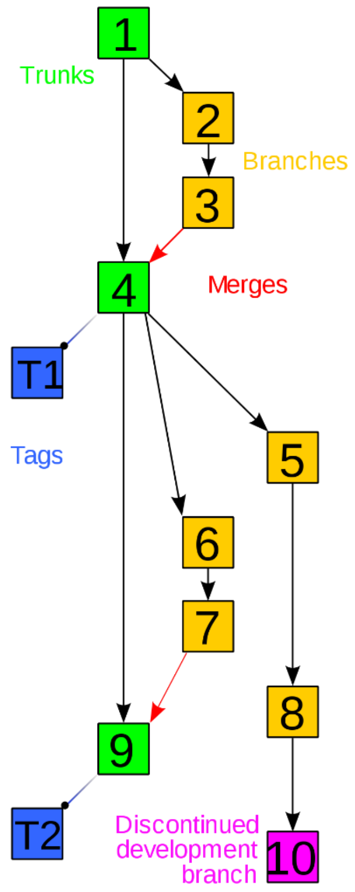
Outline

- Version Control System
- Git
- GitHub
- GitHub flow

VERSION CONTROL SYSTEM

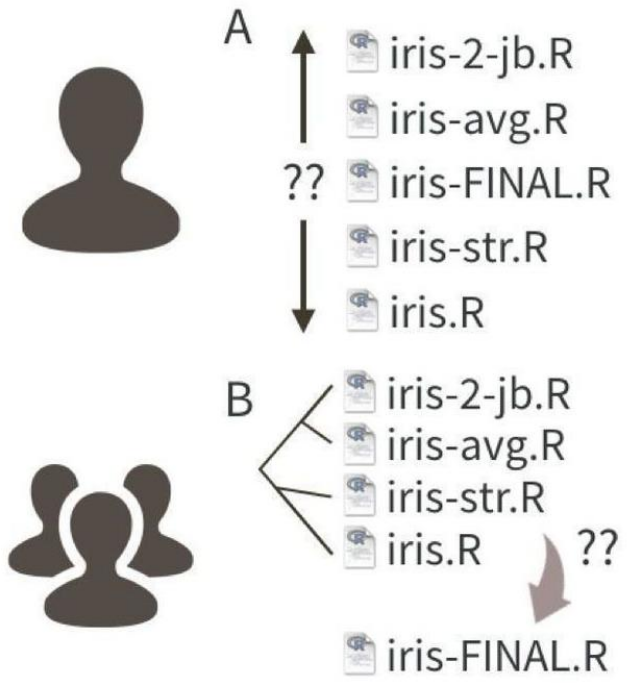
What is Version Control System (VCS)?

- VCS records changes to a file or a set of files over time
  - E.g., documents, computer programs, large web sites, and other collections of information
- It allows you to
  - **revert** the selected files back **to a previous state**,
  - revert the entire project back to a previous state,
  - **compare changes** over time, and
  - see **who last modified** something that might be causing a problem, and who introduced an issue and when, and more



Why VCS?

- If we use **different filenames** to control file versions, the names make everyone confused about relation between versions
- Things get more complex when it is a collaborative work



https://peerj.com/preprints/3159/?utm_content=bufferc4f11&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer

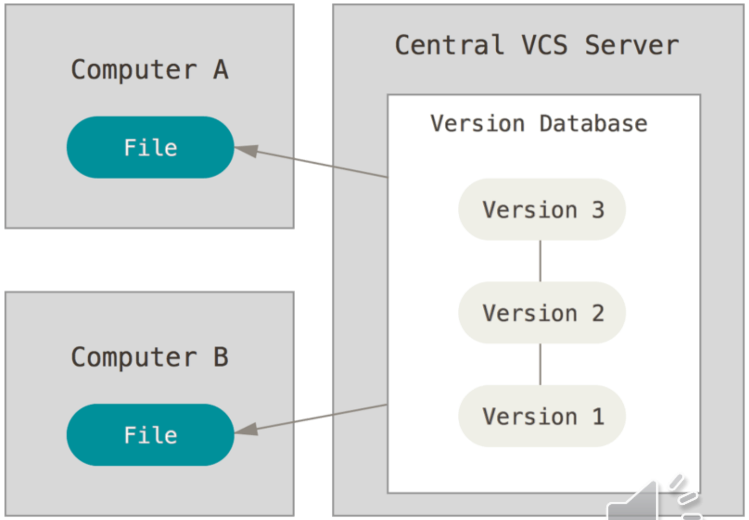
Version Control Systems

- RCS (Revision Control System), 1982
- CVS (Concurrent Versions System), 1990
- SVN (Apache Subversion), 2000
- Git, 2005

Centralized version control

These systems have:
- **a single server** that contains all the versioned files, and
- **several clients** that check out files from that central place
- For many years, this has been the standard for version control

Centralized version control



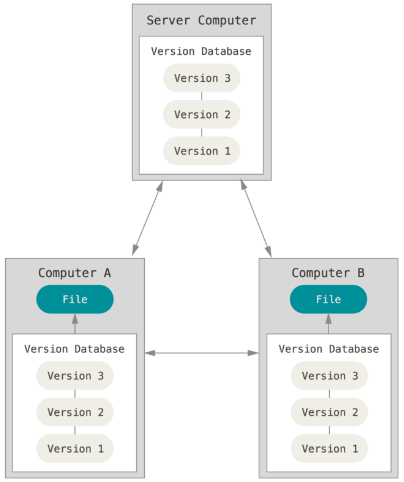
Git overview: distributed version control

Snapshot-based version control

File states

Git commands (config, init, add, commit, checkout, branch, reset)

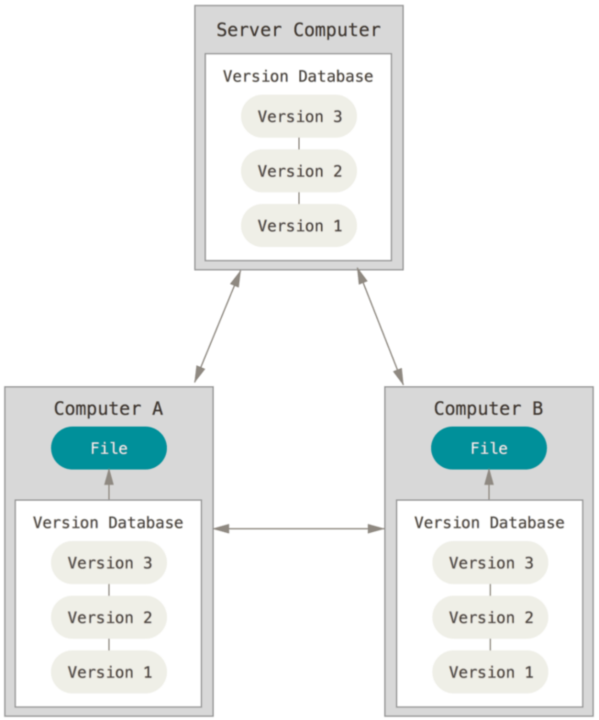
GIT



Git

- Created by Linus Torvalds in 2005 for development of the **Linux kernel**
  - with other kernel developers contributing to its initial development
- It is free and open source: https://git.kernel.org/pub/scm/git/git.git/
- Written in C for speed and portability
- **Distributed and Decentralized**
  - Independent of network access or a central server
  - **git log** is 100x faster than **svn log** because the latter must contact a remote server

Distributed version control



Git (Cont.)

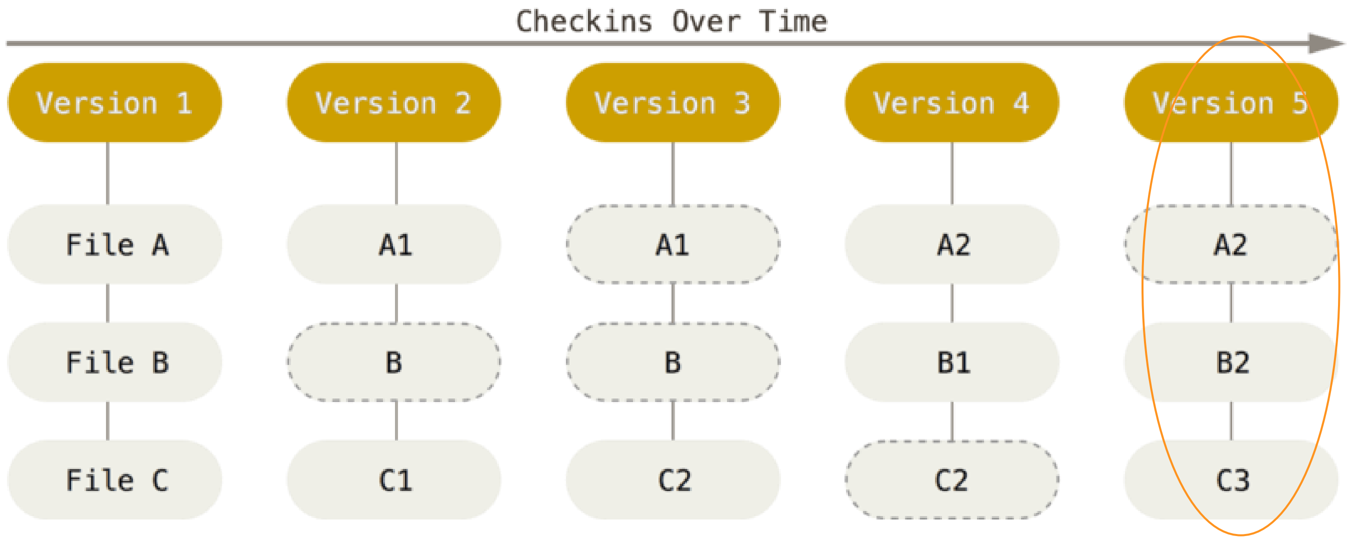
- Strong support for non-linear development
  - branching and merging
- Efficient handling of large projects
  - fetching version history from a locally stored repository faster than from the remote server

Snapshot-based Version Control

- Git considers the file contents as **a stream of snapshots**
  - A snapshot of a file is created when the file is modified
  - To be efficient, if a file is not modified, it uses a **link** to the previous version of file that has already been stored
- As a result, each version of the project contains snapshots of the monitored files

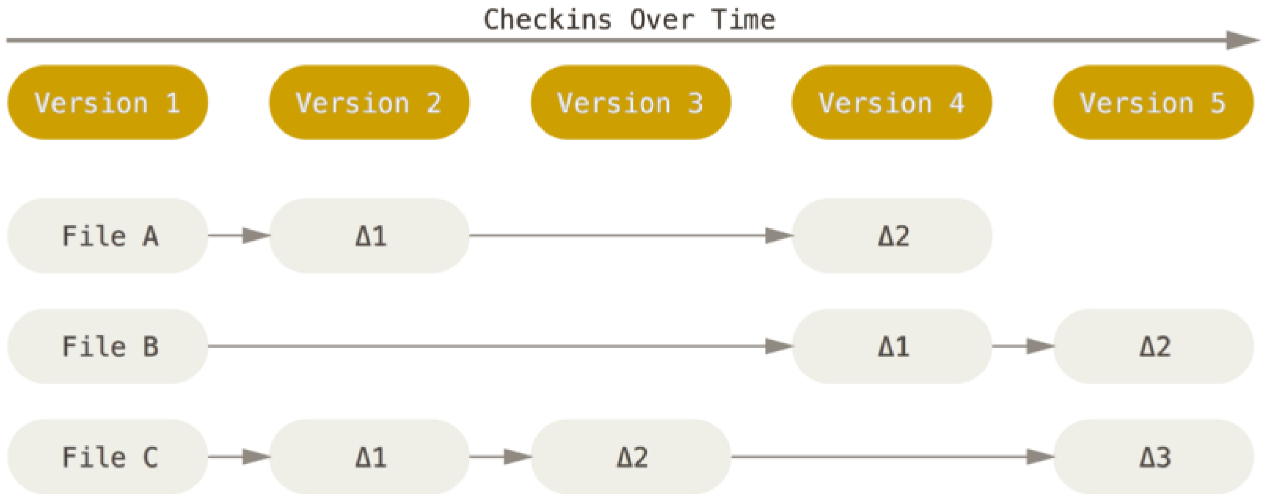
Modified, new store

Not changed, **link** to previous



Delta-based Version Control

- The version control systems, such as CVS & Subversion, keep the ***changes*** (i.e., *deltas*) made to each monitored file over time
  - By recording only the changes made to a file, it saves disk spaces
  - Recovering the file is done by **applying the series of changes made to the base version**



Check if a File Has Been Changed

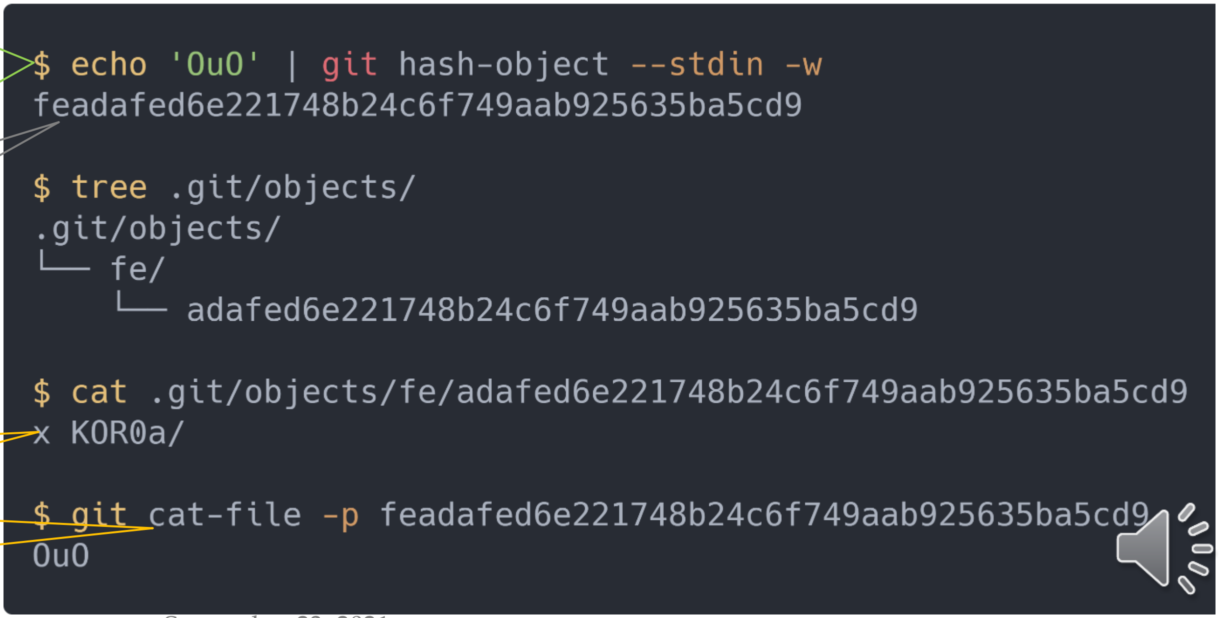
- Git uses the checksum of a file to tell if the file has been modified since the checksum has been calculated
  - The checksum is calculated by computing the **SHA-1 hash** value of **the contents of the file** or **the directory structure**
  - The SHA-1 hash value is a 40-character string composed of hexadecimal characters

git hash-object takes some data, stores it in your **.git/objects** directory (the object database), and gives you back the unique key that now refers to that data object

The first two characters of the hashing string is used as the folder name of the created object

zlib-compressed contents

To check the original content



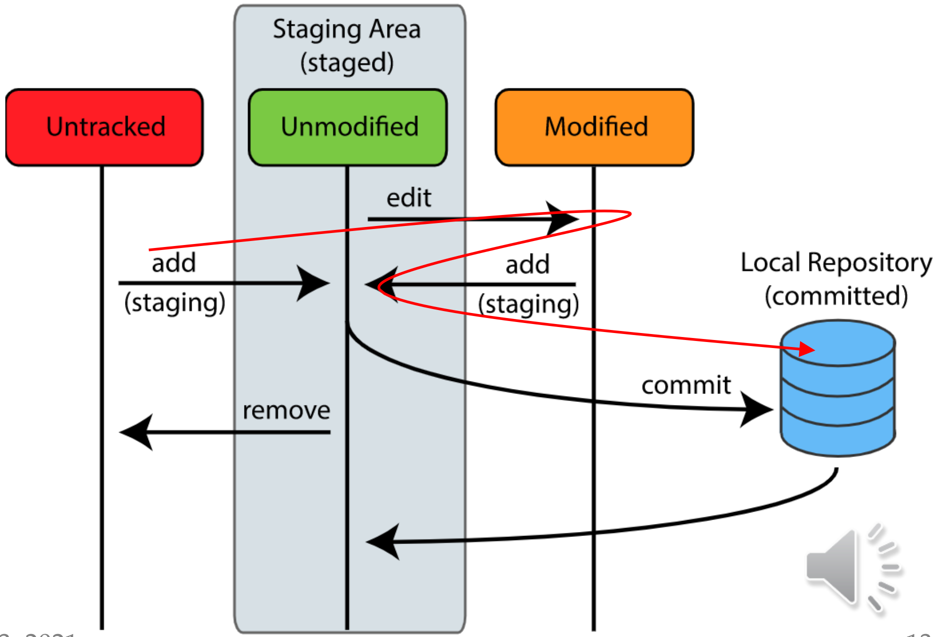
```
$ echo 'OuO' | git hash-object --stdin -w
feadafed6e221748b24c6f749aab925635ba5cd9

$ tree .git/objects/
.git/objects/
└── fe/
    └── adafed6e221748b24c6f749aab925635ba5cd9

$ cat .git/objects/fe/adafed6e221748b24c6f749aab925635ba5cd9
x KOR0a/

$ git cat-file -p feadafed6e221748b24c6f749aab925635ba5cd9
OuO
```

Courtesy of https://git-scm.com/book/en/v2/Git-Internals-Git-Objects; image source

The States of a File

- Untracked / Modified
  - Untracked means that the file is new to the Git project and **is not monitored by the versioning system**
  - Modified means that the file has been seen before (tracked) and has been changed, but **is not ready to be snapshotted by Git** (not staged)
- Staged
  - When a file becomes staged, it's taken into the staging area
  - This is where Git is able to take a snapshot of it and store its current state to the local repository
  - This area is also known as the Index
- Committed
  - Git has officially taken a snapshot of the files in the staging area and stored a unique index in the Git directory

The typical flow of manipulating a file:
Edit/Stage/Commit



Courtesy of https://code.snipcademy.com/tutorials/git/fundamentals/three-states-areas
https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things

Commands of File States

- List the states of project files with **git status**
  - It simply shows you what's been going on with **git add** and **git commit**

```
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
#modified: hello.py
#
# Changes not staged for commit:
# (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#modified: main.py
#
# Untracked files:
# (use "git add <file>..." to include in what will be committed) #
#hello.pyc
```

- To intentionally ignore the untracked files with **.gitignore**
- *A **.gitignore** file* specifies intentionally untracked files that Git should ignore
  - Files already tracked by Git are not affected
  - The example contents of *the file*

```
### C ###
# Prerequisites
*.d

# Object files
*.o
*.ko
*.obj
*.elf
*.pyc
```

Prevent compiled Python modules from appearing in **git status**

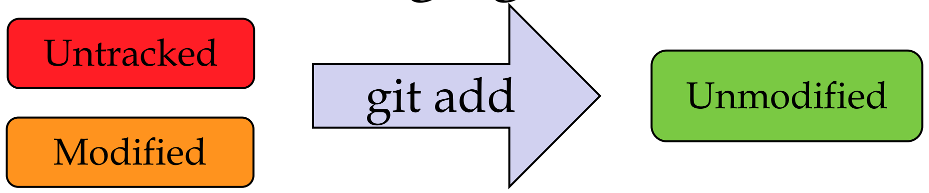
Courtesy of https://git-scm.com/docs/gitignore
https://www.atlassian.com/git/tutorials/inspecting-a-repository

Useful Commands

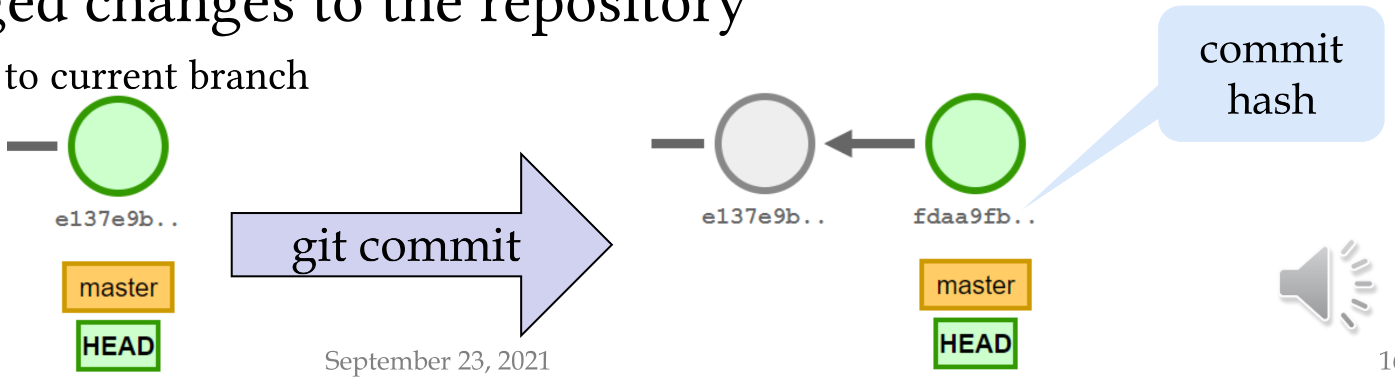
- A convenience function **git config**, which is used to set Git configuration values
  - E.g., email, username, editor and merge tool
- To set the Git configuration on different level
  - *local*: Setting is applied to the context repository **git config** gets invoked in; i.e., in the repo's **.git** directory: **.git/config**
  - *global*: Setting is applied to an operating system user in a file that is located in a user's home directory, e.g., **~ /.gitconfig**
  - *system*: Setting is applied across an entire machine, and the file is at the system root path, e.g., **$(prefix)/etc/gitconfig**
  - An example to set the user email: **git config --global user.email your_email@example.com"**

Useful Commands (Cont.)

- **git init**
  - Create an empty Git repository or reinitialize an existing one
- **git add**
  - Add file contents to the staging area

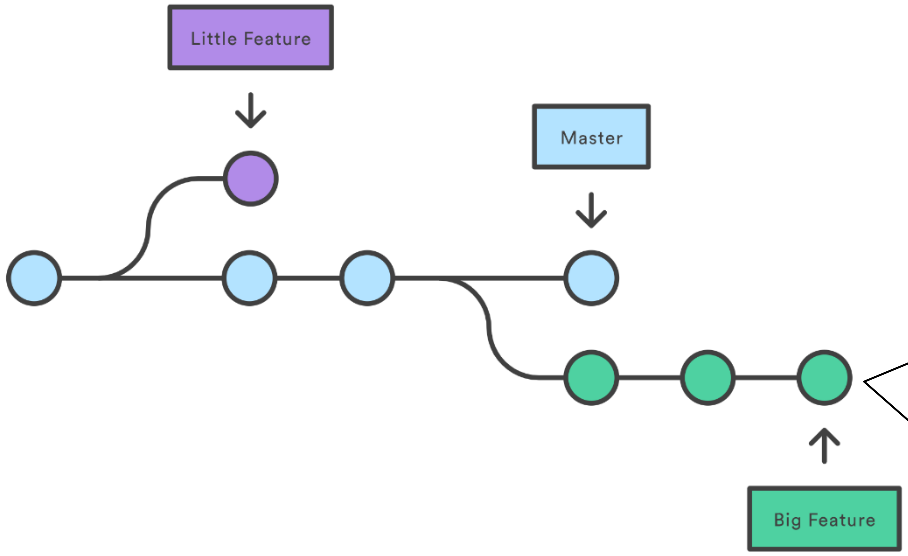


- **git commit**
  - Record staged changes to the repository
    - **HEAD** points to current branch



Useful Commands (Cont.)

- **git branch**
- Branch:
  - You spawn **a new branch** to encapsulate your changes; e.g., add a new feature or fix a bug—no matter how big or how small
  - Git branches are effectively pointers to snapshots of your changes



- The diagram visualizes a repository with **two isolated lines of development**, one for a little feature, and one for a longer-running feature
- By developing them in branches, it's not only possible to work on both of them in parallel, but it also keeps the main master branch free from questionable code

Courtesy of https://www.atlassian.com/git/tutorials/using-branches

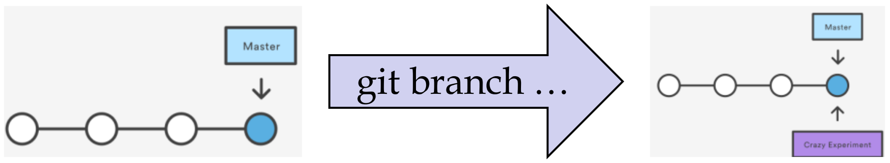
Useful Commands (Cont.)

- **git branch**
  - List all of the branches in your repository
  - This is synonymous with **git branch –list**

```
$> git branch
master another_branch
feature_inprogress_branch
```

- **git branch -a**
  - List all remote branches
- **git branch -m <branch>**
  - Rename the current branch to <branch>
- **git branch -d <branch>**
  - Delete the specified branch
  - This is a "safe" operation in that Git prevents you from deleting the branch if it has unmerged changes
  - Check **-D** option by yourself

- Example:
  - **git branch crazy-experiment**

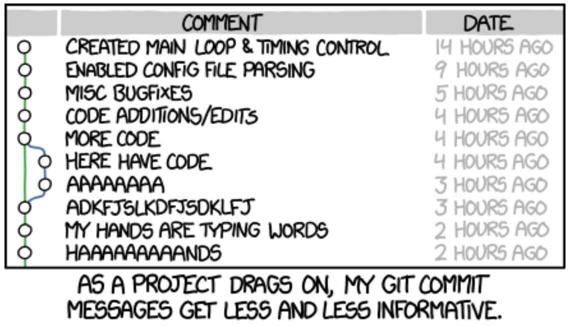


    - It's important to understand that branches are just pointers to commits
    - When you create a branch, all Git needs to do is **creating a new pointer**, it doesn't change the repository in any other way
  - This only *creates* the new branch
  - To start adding commits to it, you need to select it with **git checkout**, and then use the standard **git add** and **git commit** commands

A Good Commit Message Matters

- Communicate context about the code change to fellow developers (and indeed to their future selves)
  - A **diff** will tell you **what changed**, but only the **commit message** can properly tell you **why**

An uninformative example



	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE	4 HOURS AGO
○	AAAAAAAA	3 HOURS AGO
○	ADKFJSLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAAAAAAAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

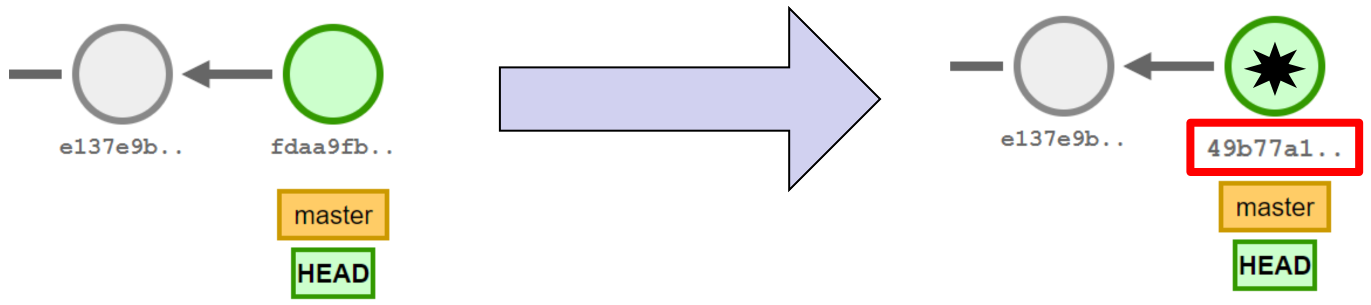
- The seven rules
  1. Separate subject from body with a blank line
  2. Limit the subject line to 50 characters
  3. Capitalize the subject line
  4. Do not end the subject line with a period
  5. Use the imperative mood in the subject line
  6. Wrap the body at 72 characters
  7. Use the body to explain what and why vs. how

An example for your reference

```
$ git log
commit 42e769bdf4894310333942ffc5a15151222a87be
Author: Kevin Flynn <kevin@flynnsarcade.com>
Date: Fri Jan 01 00:00:00 1982 -0200

Derezz the master control program

MCP turned out to be evil and had become intent on
world domination. This commit throws Tron's disc into
MCP (causing its deresolution) and turns it back into a
chess game.
```

Scenario 1
Modify the Most Recent Commit

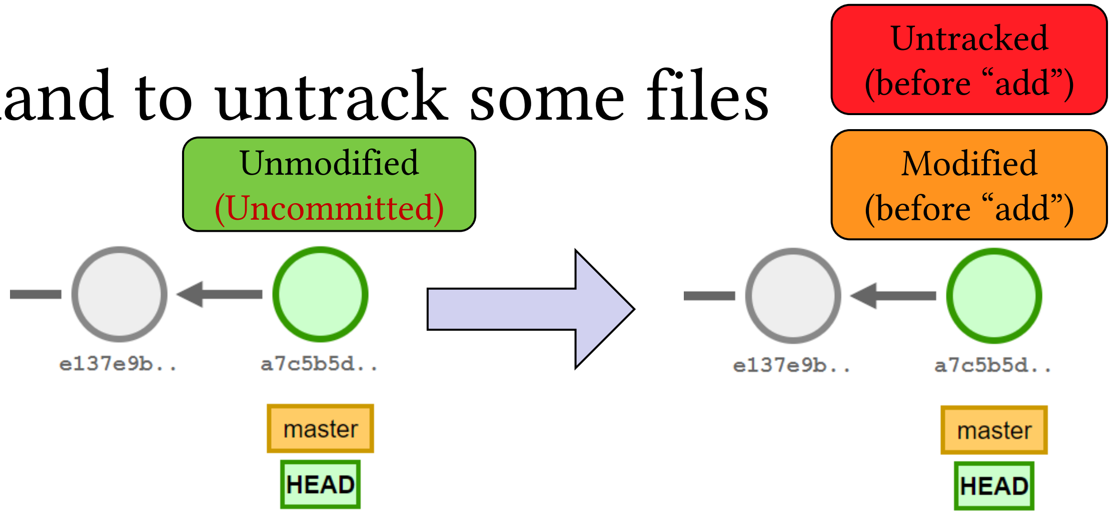
- git commit --amend
  - A convenient way to modify the most recent commit
  - It lets you combine staged changes with the previous commit instead of creating an entirely new commit
  - It can also be used to rewrite to the last commit message



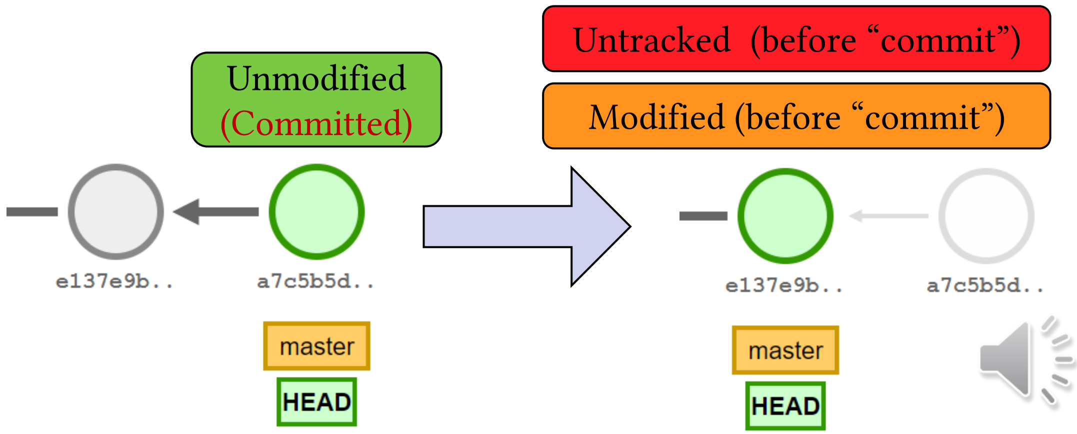
Courtesy of https://www.atlassian.com/git/tutorials/rewriting-history

Scenario 2
Undo Things

- Undo the stage command to untrack some files
  - git reset HEAD
  - git reset master
  - git reset <current SHA>

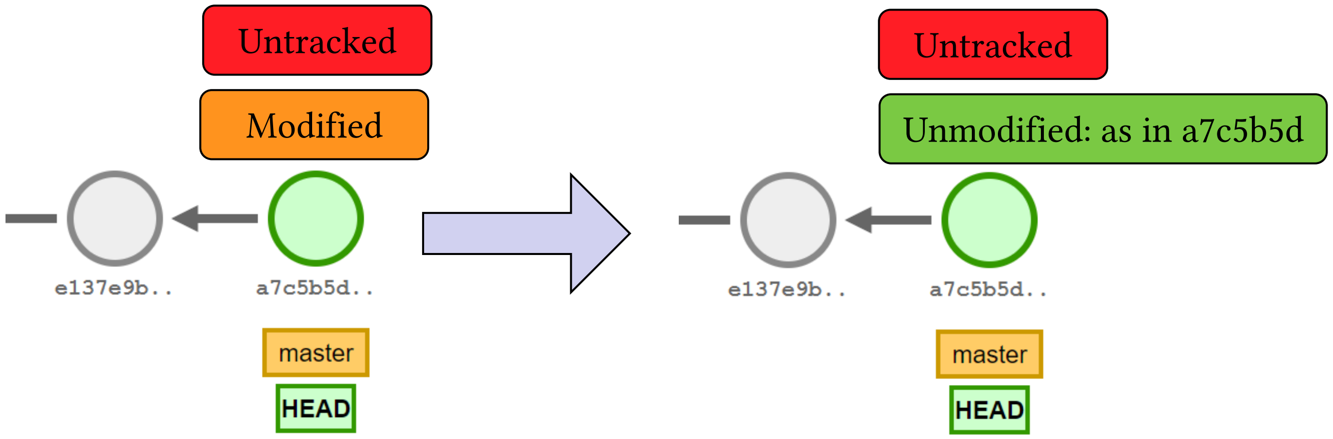


- Uncommit the last commit without recovering the files
  - git reset HEAD^
  - git reset master^
  - git reset <last SHA>



Scenario 3 Reset the Changes

- Reset to last (specific) commit
  - I.e., it destroys the modifications of the current commit
  - git reset --hard HEAD^
  - git reset --hard master^
  - git reset --hard <SHA>



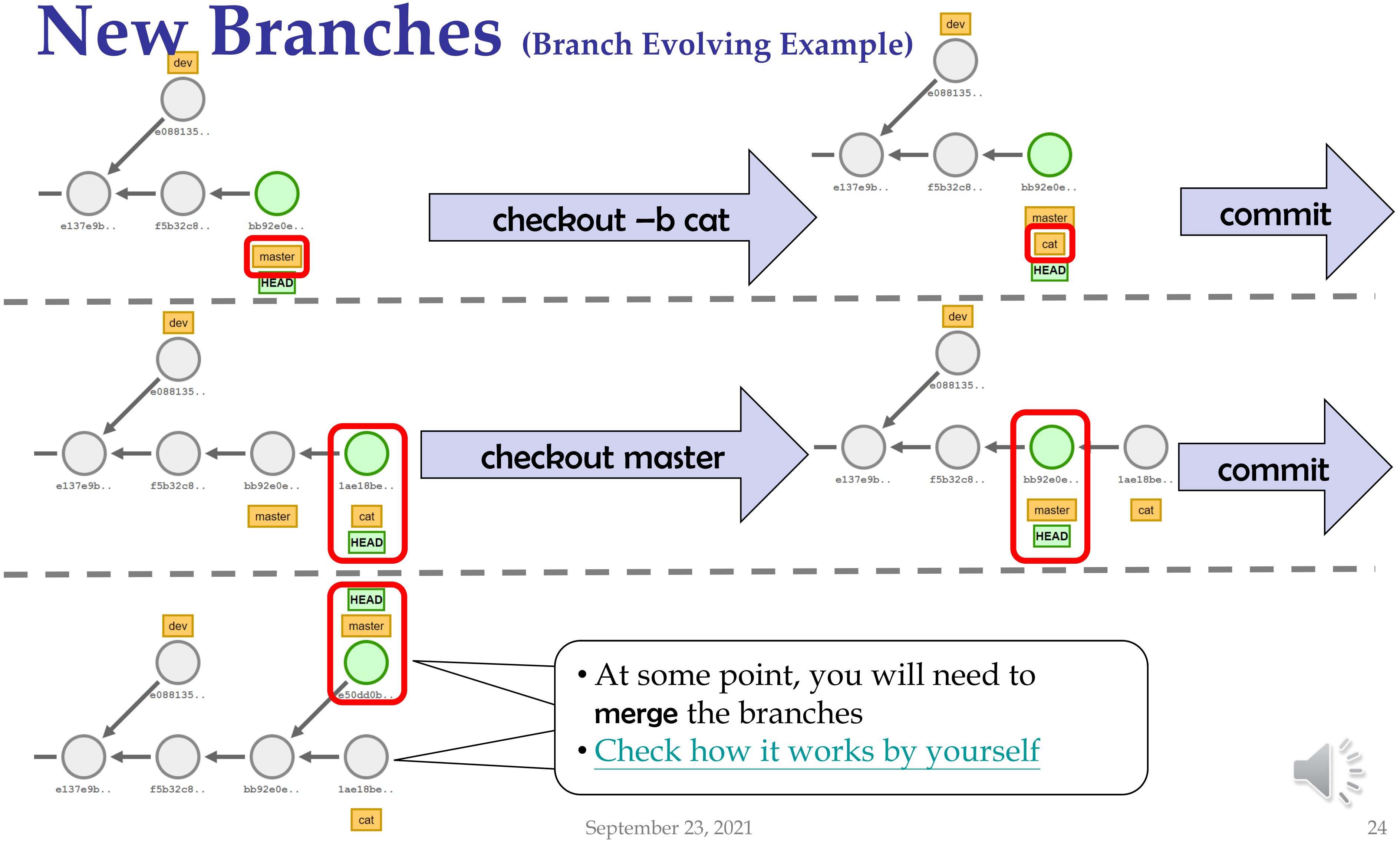
Scenario 4
New Branches

- **git checkout <branch name>**
  - switch to the specific branch, **<branch name>**
  - The **git branch** command creates a new branch off **master** using **git branch new_branch**
  - Once created, you use **git checkout new_branch** to switch to that branch

- **git checkout –b <branch name>**
  - create the new branch and immediately switch to it

- **git checkout -b <new-branch> <existing-branch>**
  - By default **git checkout -b** will base the new-branch off the current **HEAD**
  - An optional additional branch parameter can be passed to **git checkout**
  - In the above example, **<existing-branch>** is passed which then bases new-branch off of the existing-branch, instead of the current **HEAD**

Scenario 4
New Branches (Branch Evolving Example)

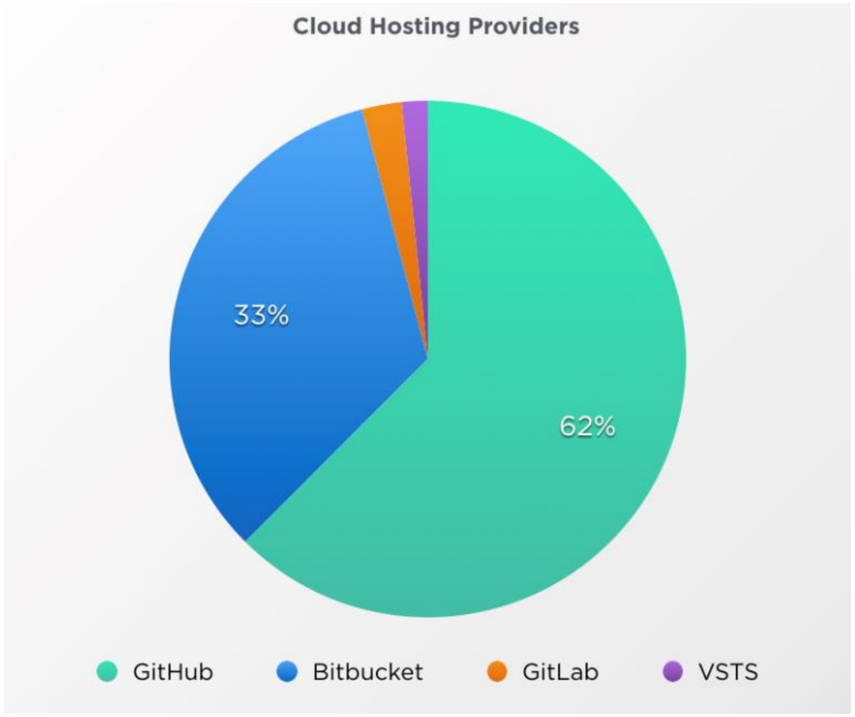


- At some point, you will need to **merge** the branches
- Check how it works by yourself

GITHUB

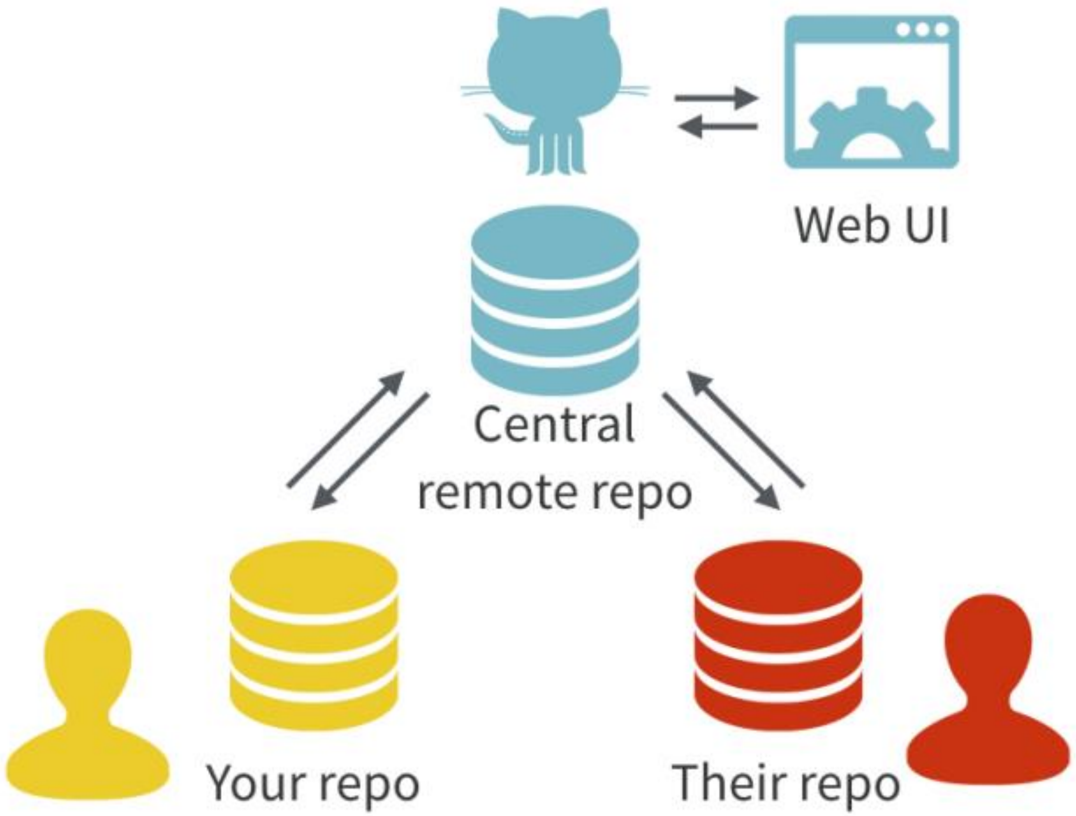
Famous Git Repository Services

- GitHub (not opensource, belong to Microsoft)
- Bitbucket
- GitLab (opensource)



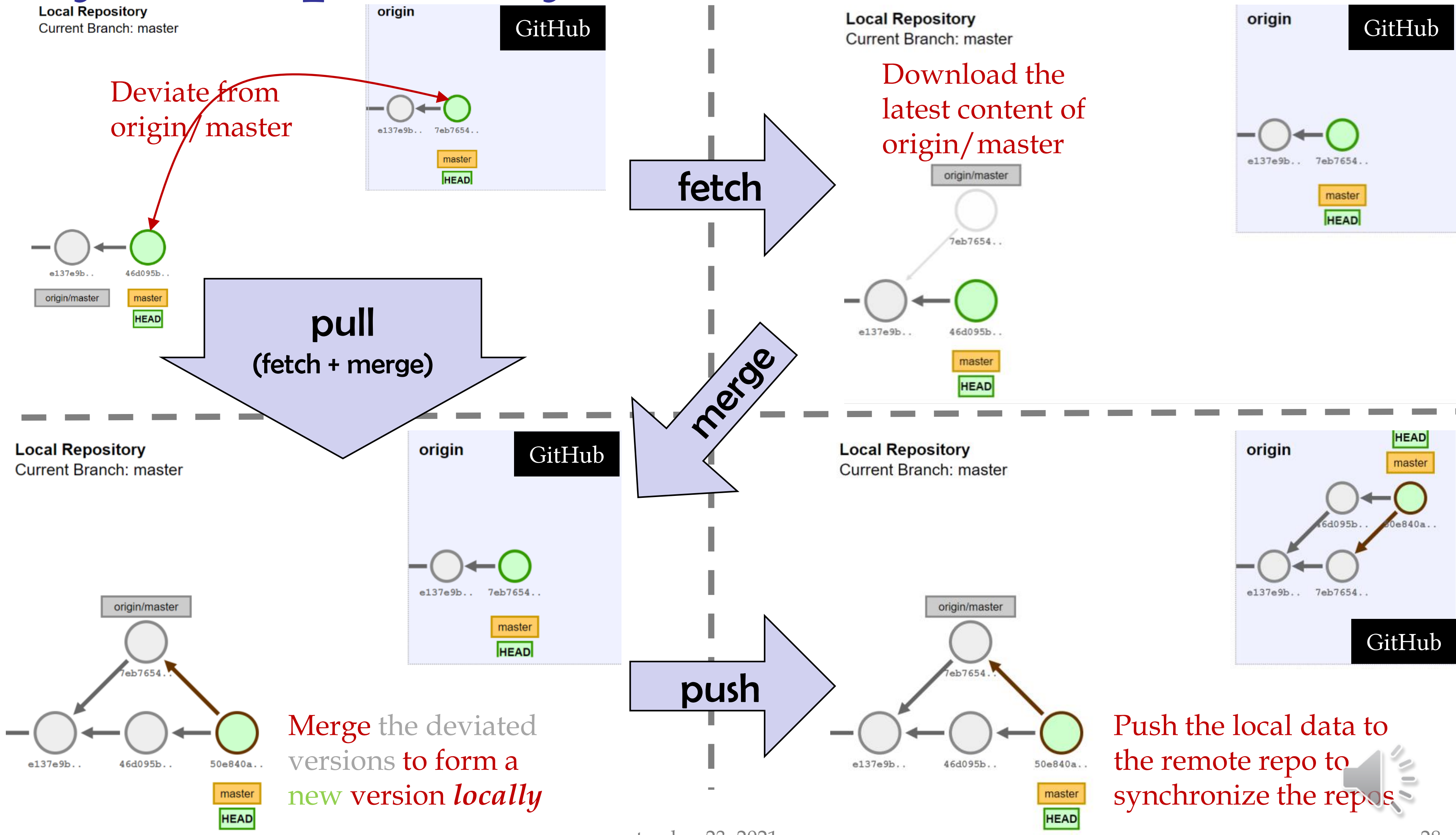
Courtesy of https://blog.bitrise.io/state-of-app-development-2016

Host Your Project on GitHub



Courtesy of
https://peerj.com/preprints/3159/?utm_content=bufferc4f11&utm_medium=social&utm_source=twitter.com&utm_campaign=buffer

Sync Repository with GitHub



Deviate from origin/master

Download the latest content of origin/master

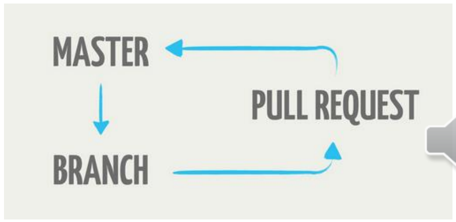
Merge the deviated versions to form a new version *locally*

Push the local data to the remote repo to synchronize the repos

GitHub Flow

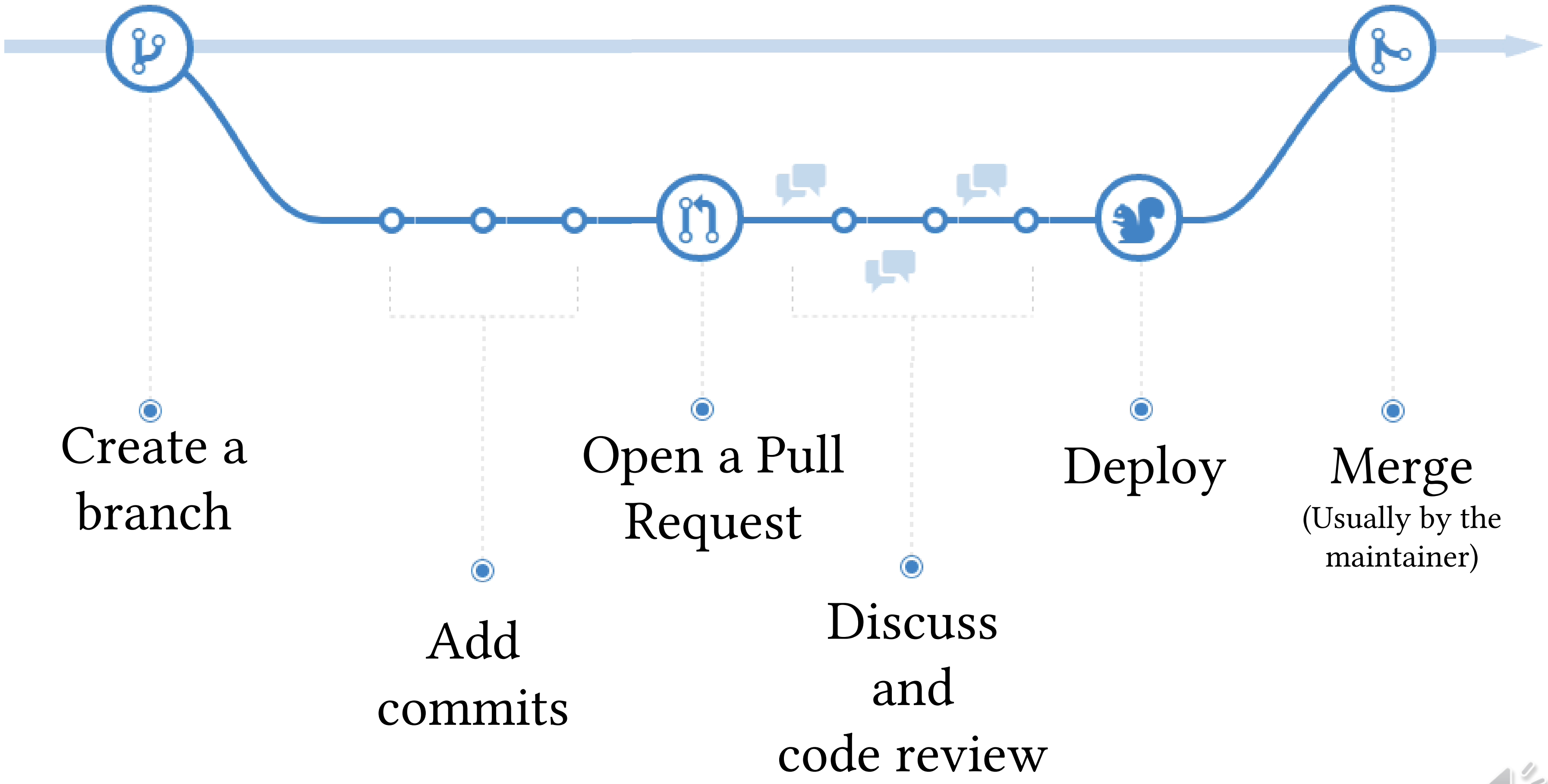
- GitHub flow is a lightweight, branch-based workflow that supports teams and projects

- A typical flow
  - Create a branch from **master** for new features
  - After added the features with some commits, you **push** them to GitHub as same branch
  - Open a ***pull request*** for adding the new branch from the **master** branch



Courtesy of https://guides.github.com/introduction/flow/

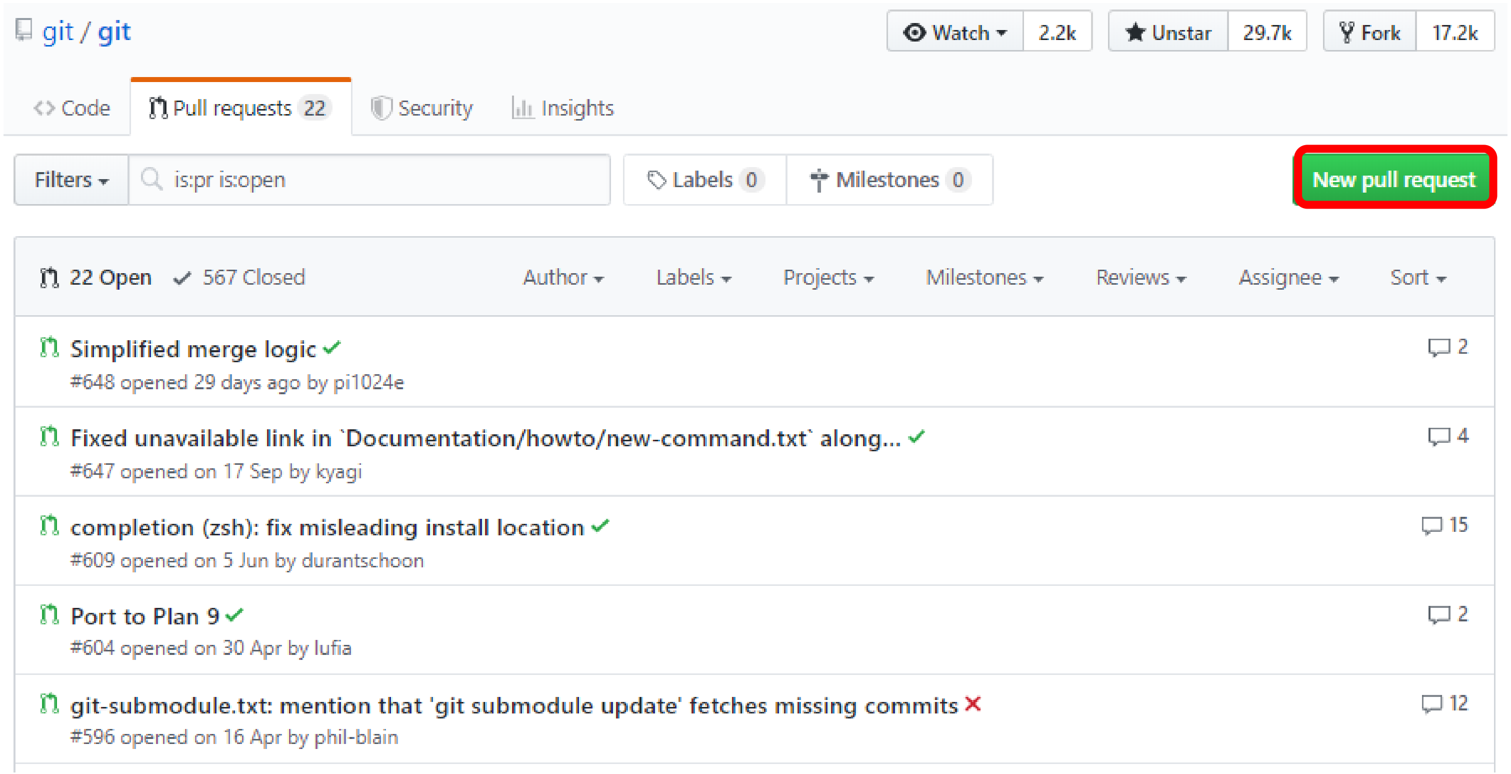
GitHub Flow (Cont.)



Courtesy of https://guides.github.com/introduction/flow/

Pull Request

- Usually provided by the Git repository service
- Web UI



Simplified merge logic ✓ — 2
#648 opened 29 days ago by pi1024e

Fixed unavailable link in `Documentation/howto/new-command.txt` along... ✓ — 4
#647 opened on 17 Sep by kyagi

completion (zsh): fix misleading install location ✓ — 15
#609 opened on 5 Jun by durantschoon

Port to Plan 9 ✓ — 2
#604 opened on 30 Apr by lufia

git-submodule.txt: mention that 'git submodule update' fetches missing commits ✗ — 12
#596 opened on 16 Apr by phil-blain

Pull Request (Cont.)

- Two workflows

1. Pull Request from a **forked** repository
   - A fork is a copy of a repository used in the ***fork and pull model***
   - Anyone can fork an existing repository and push changes to their personal fork without needing access to the source repository
   - The changes can be pulled into the source repository by the project maintainer
2. Pull Request from a **branch** within a repository
   - Used in the ***shared repository model***, collaborators are granted push access to a single shared repository and topic branches are created when changes need to be made
   - Pull requests are useful in this model as they initiate code review and general discussion about a set of changes before the changes are merged into the main development branch
   - This model is more prevalent with **small teams and organizations collaborating on private projects**

Courtesy of https://help.github.com/en/github/collaborating-with-issues-and-pull-requests/about-collaborative-development-models

References

- Git user manual
- Visualizing Git Concepts with D3
- 為你自己學 Git – 高見龍 (線上電子書）