DIHSÜD Digital Innovation Hub

FH JOANNEUM University of Applied Sciences

Git Workshop

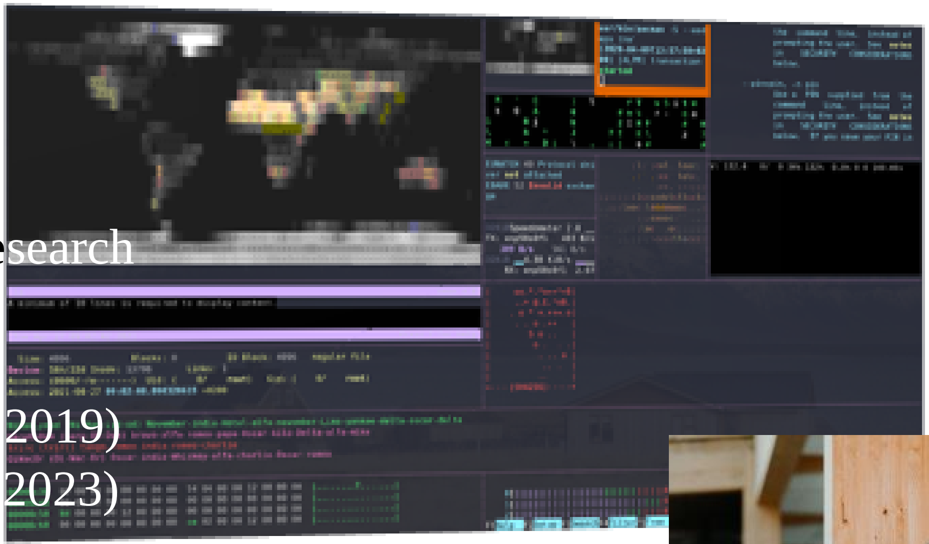
Harald Schwab, BSc MSc

27.10.2025

$whoami

Harald Schwab

- Software Engineering in Teaching and Research
- 15+ year of experience in IT
  (system administration, software development, …)
- Bachelor: ITM FH JOANNEUM (2016 - 2019)
- Master: IMS FH JOANNEUM (2019 - 2023)
- Since 2017 Staff FH JOANNEUM (Tutor, Praktika, WiMa, Lecturer)

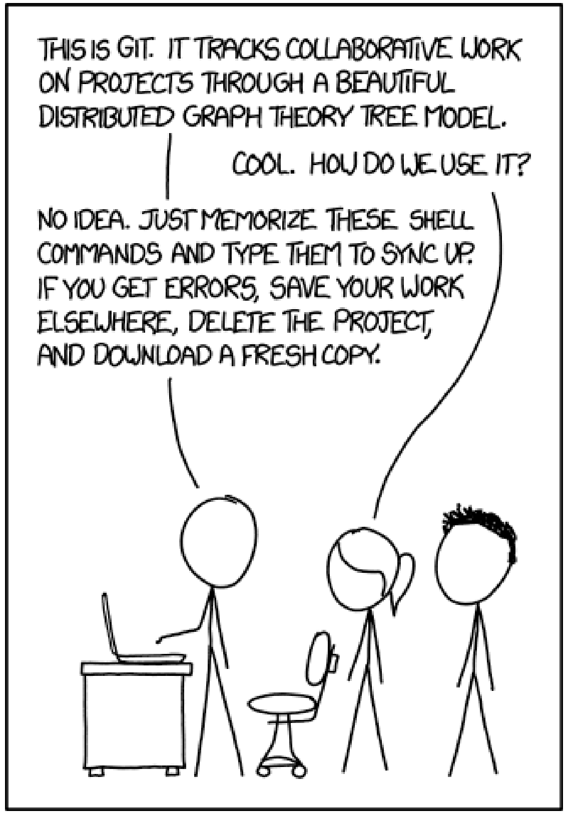


Contact:

- harald.schwab2@fh-joanneum.at

Agenda

- Version Control Systems
- What is Git
- Git Workflow
- Workspace, Staging, Commit
- .gitignore
- Branching and Merging
- Work with remote repositories

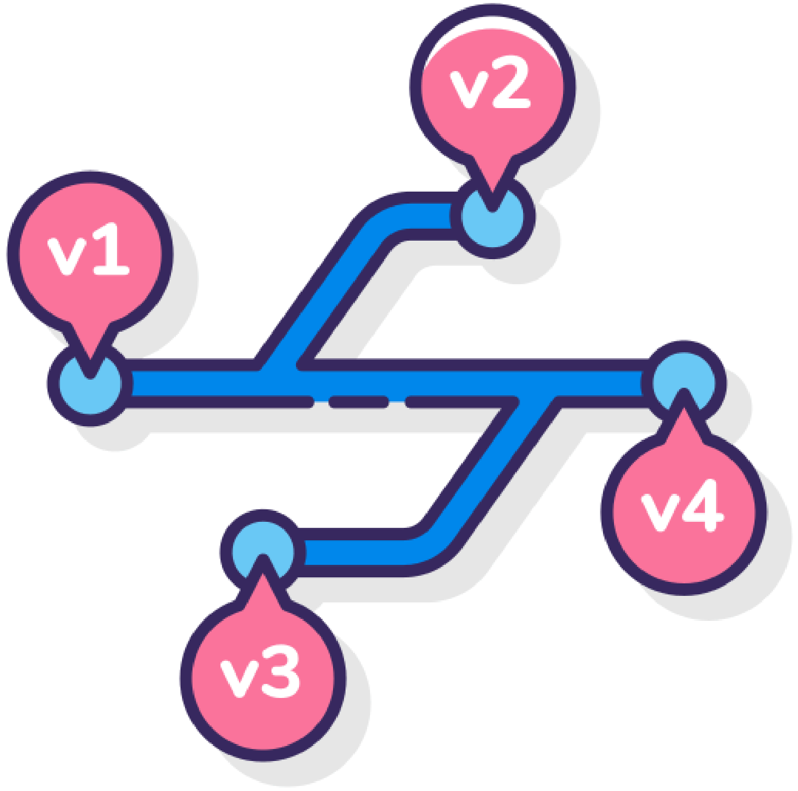

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Source: https://xkcd.com/1597/

Using Version Control (Systems)

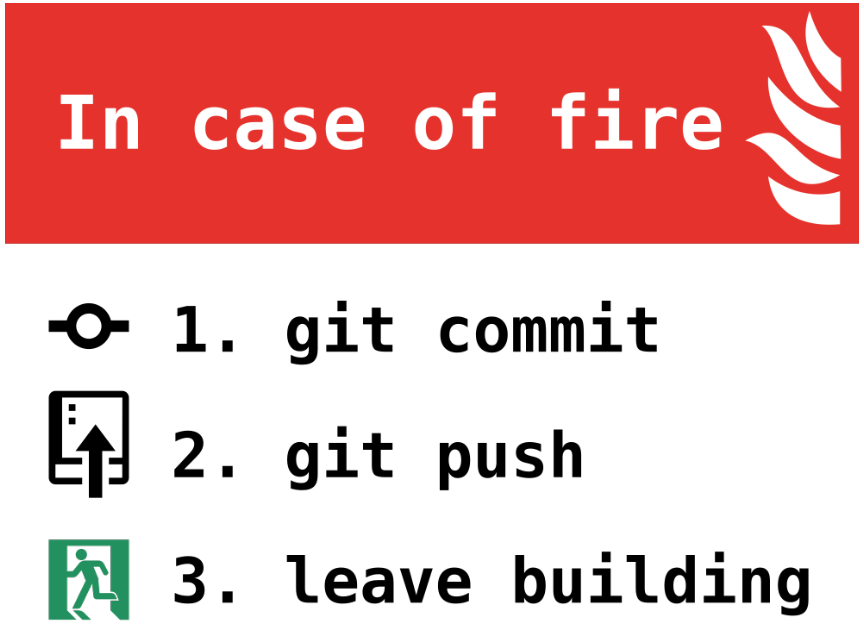
A **Version Control System (VCS)** records changes to a file or set of files over time so that you can recall specific versions later.

- Retains, and provides access to, **every version** of **every file** that has ever been stored in it.
- Provides **metadata**, like commit messages, attached to single files or collections of files.
- Allows teams that may be **distributed** across space and time to **collaborate**.



Practical use of VCS

- **Keep absolute everything[1] in the version control**
- Check in regularly
- Use **meaningful** commit messages
- Track changes



[1] *text based, that is not easily be recreated*

There are more than only Source Code Files

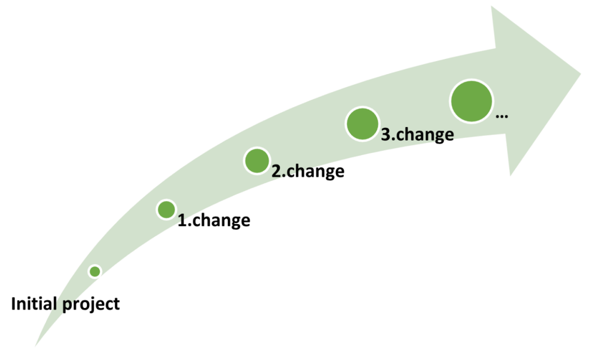
Each project contains different file types, so we need to think about how to create a structure to work with for a long time

• Java source	• *.java
• Java class	• *.class
• Java archive	• *.jar
• Configuration	• *.properties, *.xml, *.json, …
• Documentation	• *.txt, *.md, …
• Images	• *.png, *.jpg, …
• …	• …

Java is used here exclusively as an example, this applies to all common software projects.

Keep track of project changes in software development

- **Record changes** to (all) files, like *add*, *remove*, *modify*, … any content of (nearly) any file.
- Access to **specific version** over time, which you have tracked with Source Code Versioning, like **git**
- **Revert** to a previous state in case of bugs, problems, … and stop working with “comments” of deprecated source code
- **Compare changes**, to find differences between version, it’s important for debugging, bugfixing, … e.g. compare version 23 with version 54, which lines are different and will lead to a failure?
- **Track changes** and who has done those changes, in case of questions, you are able to identify current and last developers



First touchpoints in Real Life

How have you done versioning …

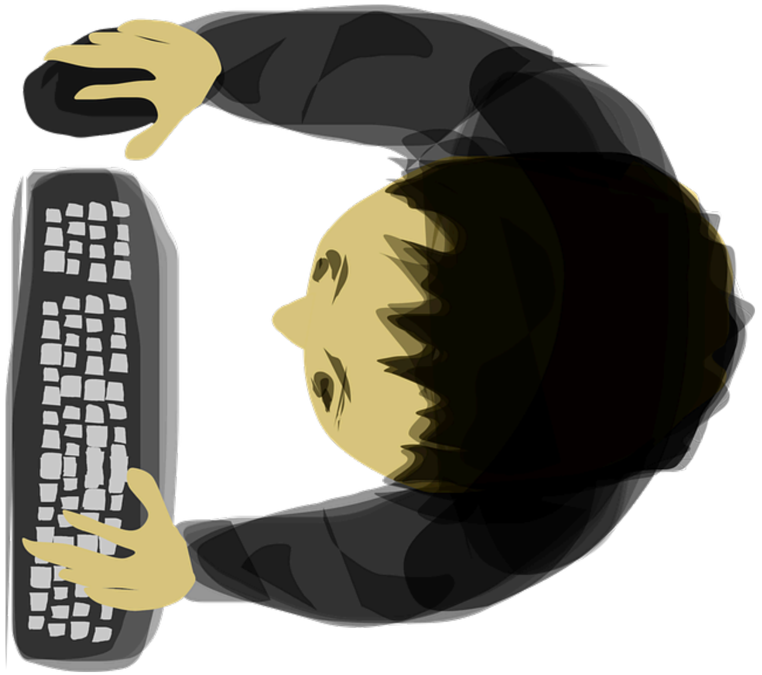
… at work?

… on a previous project?

… on homeworks?

… with paperwork?

…

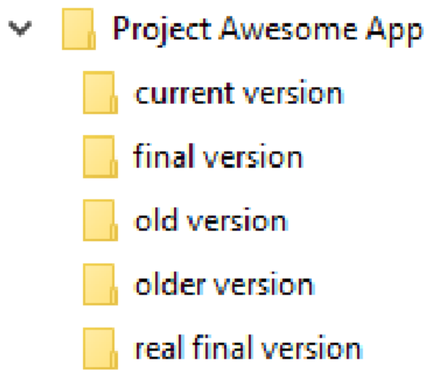


Source: www.pixabay.com

Some people's version control method until now

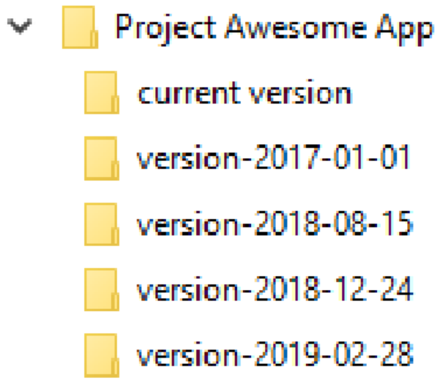
Textual version

- Descriptive
- "Quick"
- Confusing
- …



```
Project Awesome App
    current version
    final version
    old version
    older version
    real final version
```

Timestamp version



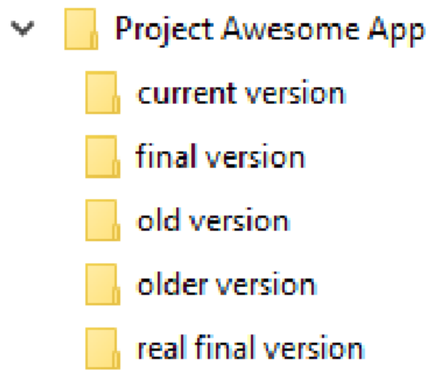
```
Project Awesome App
    current version
    version-2017-01-01
    version-2018-08-15
    version-2018-12-24
    version-2019-02-28
```

- Early approach
- "Improved"
- Confusing
- …

Both versions <u>may not</u> the best way for source code versioning (!)

Possible reasons “directory-versioning” is not a good practice

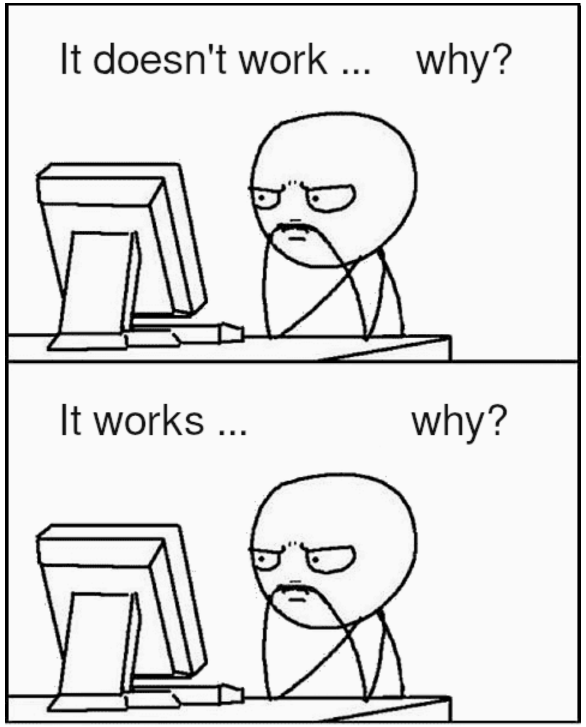
- Really **difficult to compare** old versions
- **Manual comparison** is needed and is defective
- Many other manual work needed
- No easy way for commenting your changes
- **Getting more and more complex** at longer “life-time”
  - Think about developing a software over months or years
  - 2 – 4 “versions” per week
  - Up to approximately 200 versions a year
  - … in a single folder with manual comparison
    “finally the newest version of last changes but not fully tested”



Project Awesome App
current version
final version
old version
older version
real final version

Already started?

- First line of code has been already developed
- First challenges of changes
  - It works now
  - It doesn’t work, but what I’ve changed
  - Oh… I’m not sure, there’s a backup
  - Oh… there is **no** backup[2]
  - …
- After this workshop
  - **It doesn’t work, I will be able to look on what I’ve changed!**

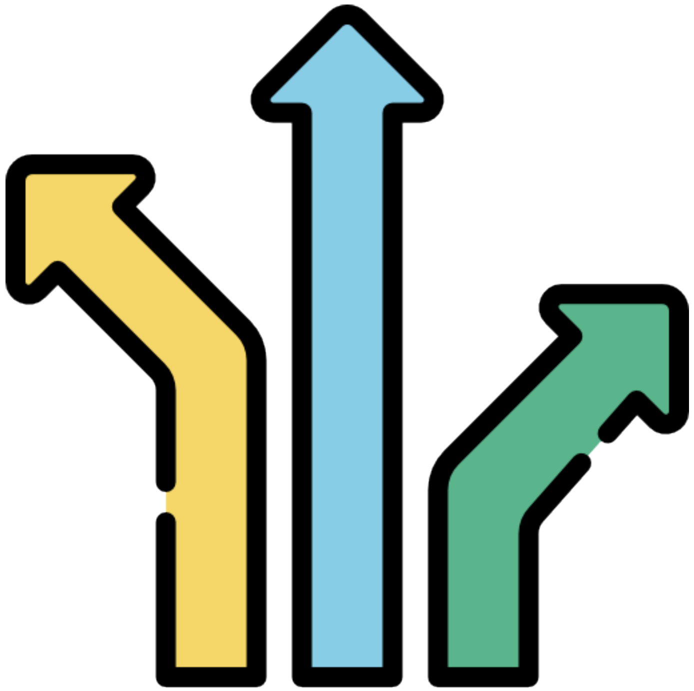


[2] *Software Versioning is no replacement of traditional backup strategies!*

Different kind of VCS

There are basically three different types of VCS

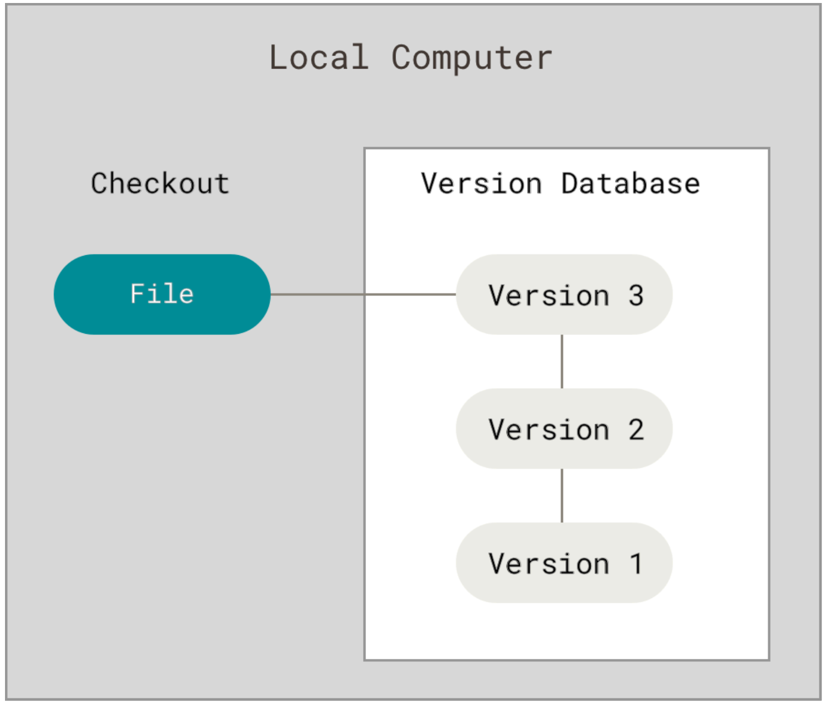
- **Local**
  - ※ only on your computer
  - ※ no access of other people
- **Centralized**
  - ※ only on a single server
  - ※ any authorized person is able to read and write changes
- **Distributed**
  - ※ each server and computer has a full copy
  - ※ any changes will be shared and stored at devs pc



Local Version Control System

You are developing (alone) on your local computer

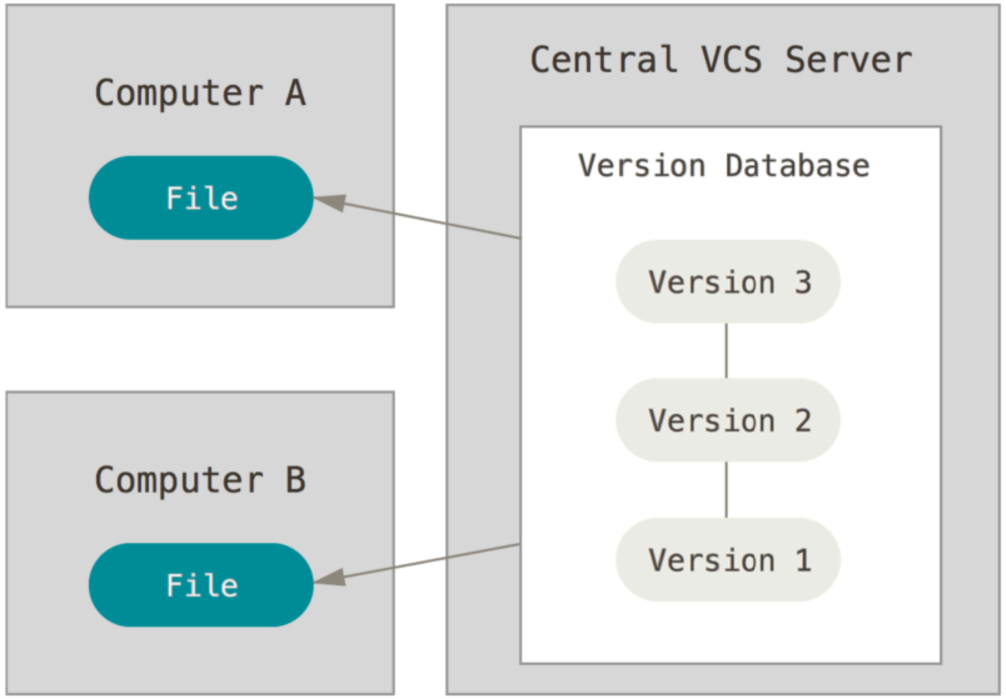
- **Version Database** is stored locally.
- Working Directory / **Checkout** is located on your local machine.
- Every change your are transmitting to your local Version Database.



Centralized Version Control System

You’re working on your local computer, other devs work on their own computer

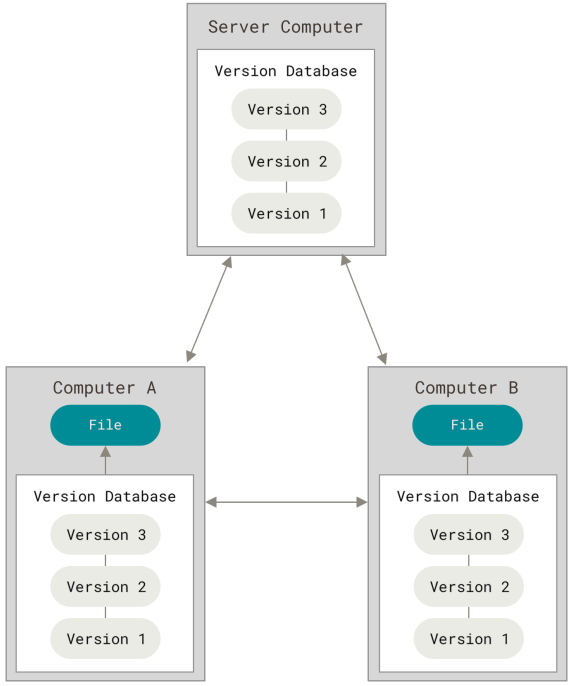
- Each developer has only the current version in his **working directory**.
- Changes will be transferred to a centralized server **repository**.
- Each developer has the possibility to keep on track of latest changes of other developers.



Distributed Version Control System

Combination of local and distributed approach with it’s (dis)advantages

- Each developer is able and has to
  - **create new versions** through committing changes to the VCS
  - **synchronizes changes** to a single or multiple servers
  - **collaborate with other team members** through a well known VSC
  - and **keep on track to all changes** within the team.



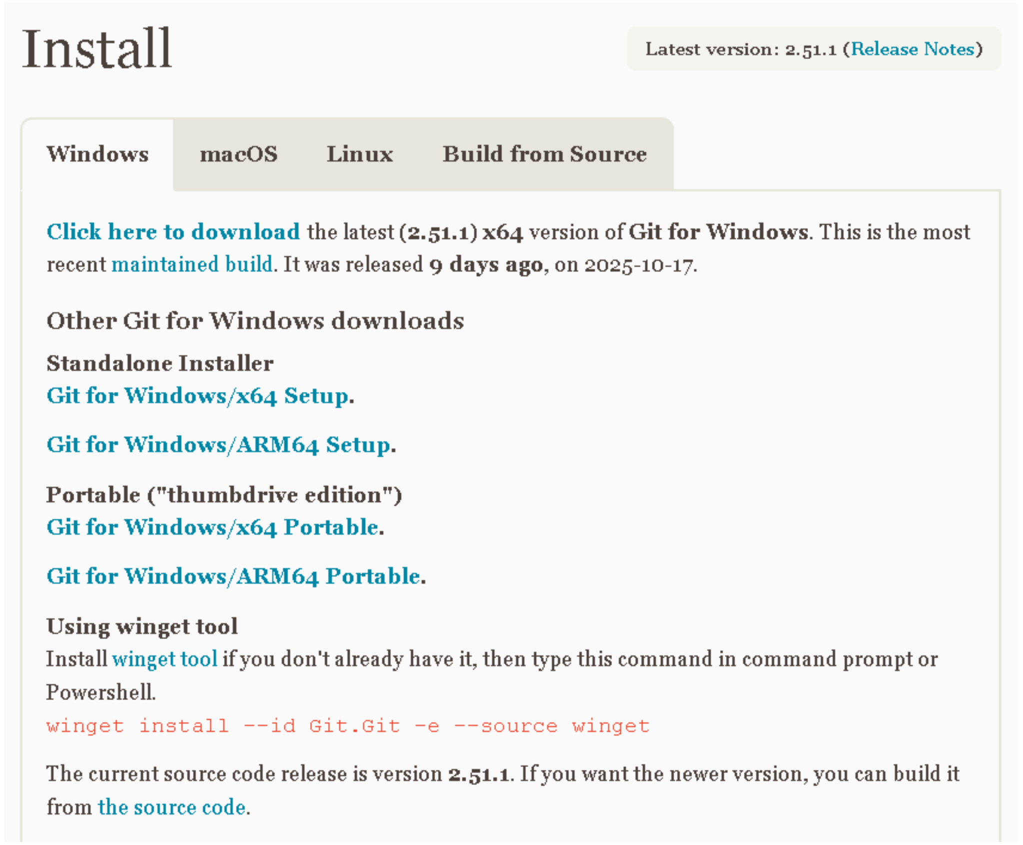
Build software better, together

- **Branching** and **Merging**
- Small and fast
- **Distributed**
- Data assurance
- **Staging area**
- Free and open source

DIHSÜD Digital Innovation Hub

FH | JOANNEUM University of Applied Sciences

Install Git – *Windows*



Install

Latest version: 2.51.1 (Release Notes)

Windows | macOS | Linux | Build from Source

Click here to download the latest (**2.51.1**) **x64** version of **Git for Windows**. This is the most recent maintained build. It was released **9 days ago**, on 2025-10-17.

Other Git for Windows downloads

Standalone Installer
Git for Windows/x64 Setup.

Git for Windows/ARM64 Setup.

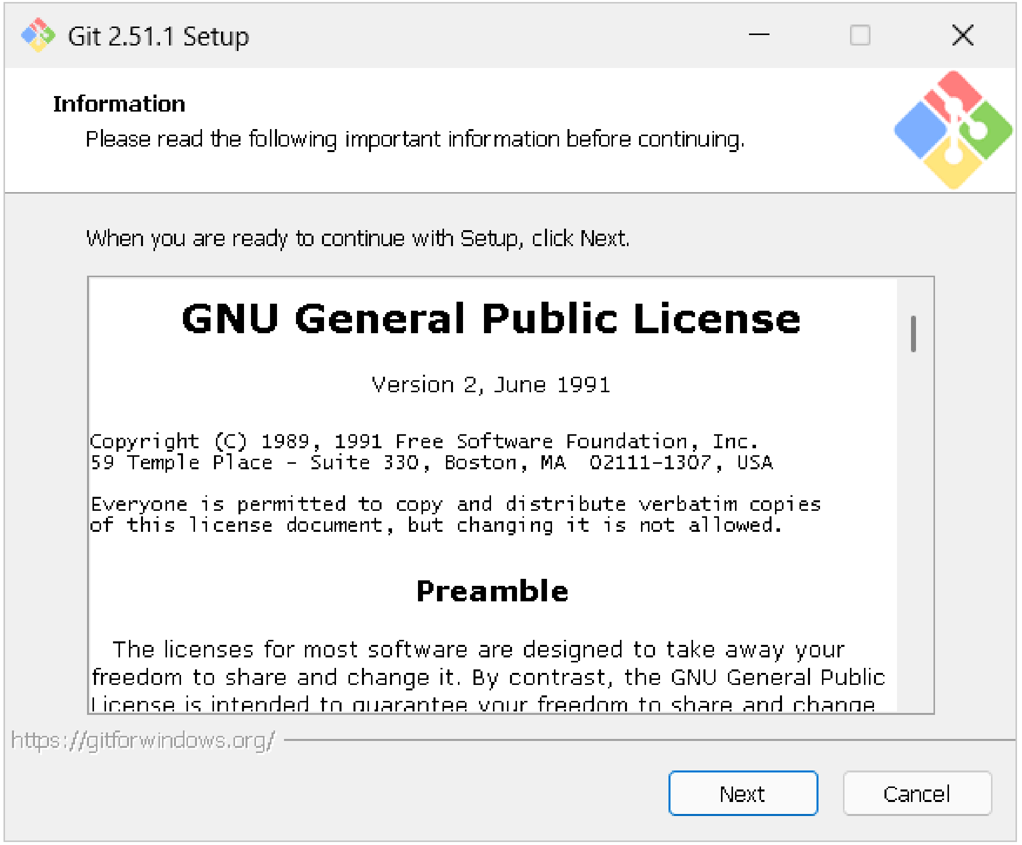
Portable ("thumbdrive edition")
Git for Windows/x64 Portable.

Git for Windows/ARM64 Portable.

Using winget tool
Install winget tool if you don't already have it, then type this command in command prompt or Powershell.

```
winget install --id Git.Git -e --source winget
```

The current source code release is version **2.51.1**. If you want the newer version, you can build it from the source code.



Git 2.51.1 Setup

Information
Please read the following important information before continuing.

When you are ready to continue with Setup, click Next.

GNU General Public License

Version 2, June 1991

```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place - Suite 330, Boston, MA  02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

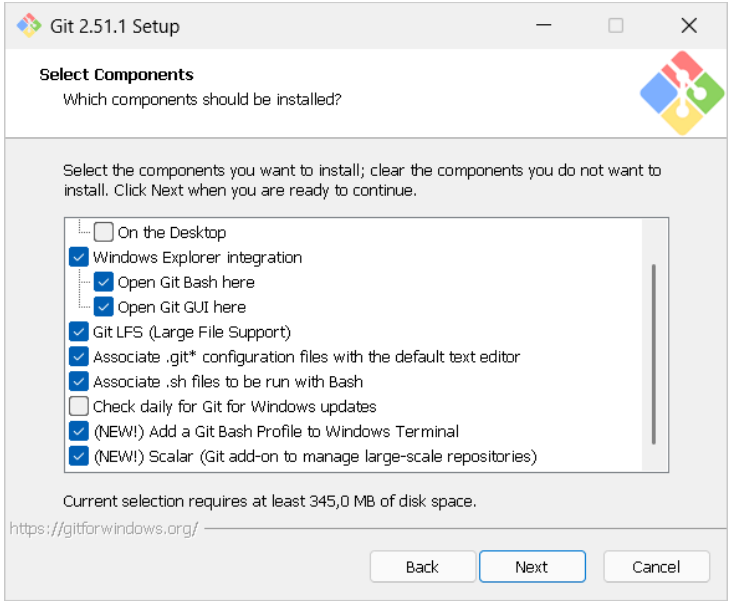
Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change

https://gitforwindows.org/

https://git-scm.com/install/windows

Install Git – *Windows*



Git 2.51.1 Setup

Select Components

Which components should be installed?

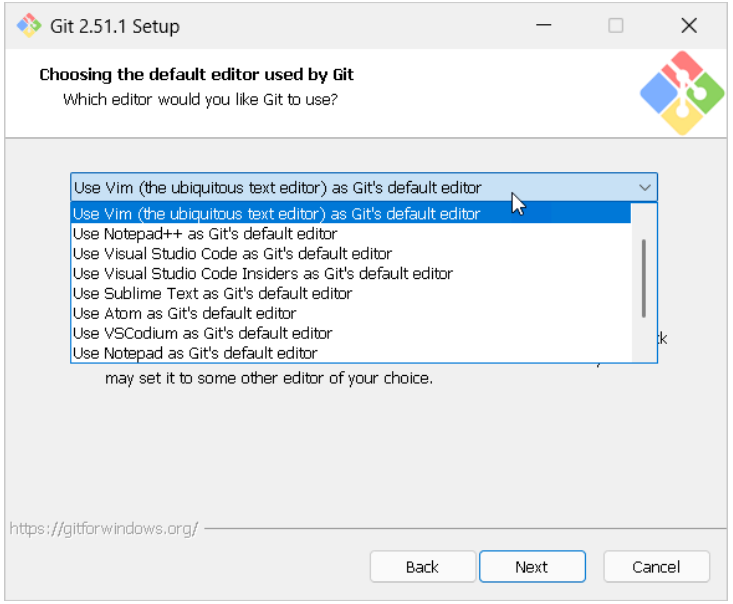
Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- [ ] On the Desktop
- [x] Windows Explorer integration
  - [x] Open Git Bash here
  - [x] Open Git GUI here
- [x] Git LFS (Large File Support)
- [x] Associate .git* configuration files with the default text editor
- [x] Associate .sh files to be run with Bash
- [ ] Check daily for Git for Windows updates
- [x] (NEW!) Add a Git Bash Profile to Windows Terminal
- [x] (NEW!) Scalar (Git add-on to manage large-scale repositories)

Current selection requires at least 345,0 MB of disk space.

https://gitforwindows.org/

If using, select the option “Add a Git Bash Profile to Windows Terminal”



Git 2.51.1 Setup

Choosing the default editor used by Git

Which editor would you like Git to use?

Use Vim (the ubiquitous text editor) as Git's default editor

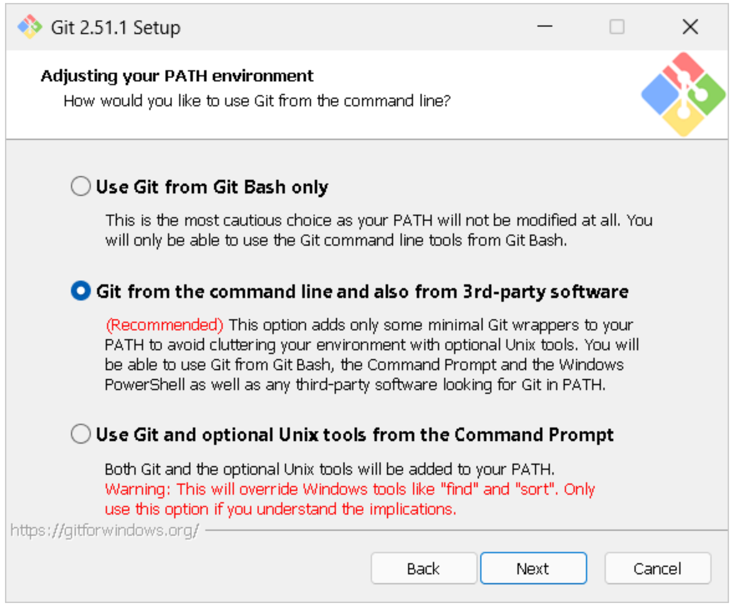
- Use Vim (the ubiquitous text editor) as Git's default editor
- Use Notepad++ as Git's default editor
- Use Visual Studio Code as Git's default editor
- Use Visual Studio Code Insiders as Git's default editor
- Use Sublime Text as Git's default editor
- Use Atom as Git's default editor
- Use VSCodium as Git's default editor
- Use Notepad as Git's default editor

may set it to some other editor of your choice.

https://gitforwindows.org/

Choose an editor for commit messages etc. The default would be **vim**, a CLI terminal

https://git-scm.com/install/windows


Git 2.51.1 Setup

Adjusting your PATH environment

How would you like to use Git from the command line?

( ) **Use Git from Git Bash only**

This is the most cautious choice as your PATH will not be modified at all. You will only be able to use the Git command line tools from Git Bash.

(●) **Git from the command line and also from 3rd-party software**

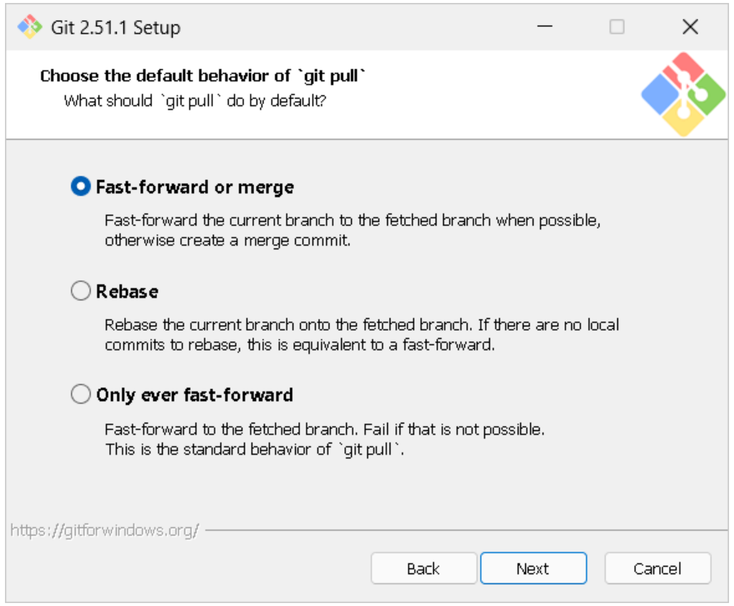
(Recommended) This option adds only some minimal Git wrappers to your PATH to avoid cluttering your environment with optional Unix tools. You will be able to use Git from Git Bash, the Command Prompt and the Windows PowerShell as well as any third-party software looking for Git in PATH.

( ) **Use Git and optional Unix tools from the Command Prompt**

Both Git and the optional Unix tools will be added to your PATH.
Warning: This will override Windows tools like "find" and "sort". Only use this option if you understand the implications.

https://gitforwindows.org/

You can select how to include Unix CLI commands, coming with `git`, like `cat`, `grep`, etc. I would recommend to keep the selected option


Git 2.51.1 Setup

Choose the default behavior of `git pull`

What should `git pull` do by default?

(●) **Fast-forward or merge**

Fast-forward the current branch to the fetched branch when possible, otherwise create a merge commit.

( ) **Rebase**

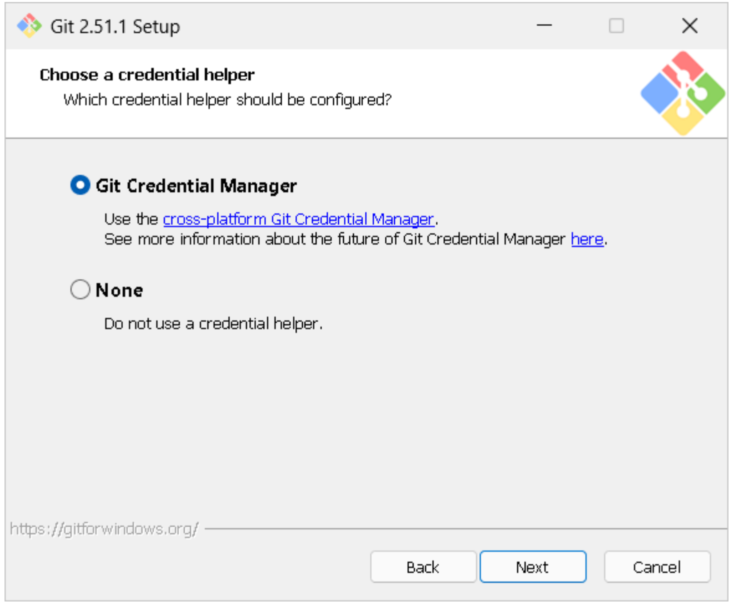
Rebase the current branch onto the fetched branch. If there are no local commits to rebase, this is equivalent to a fast-forward.

( ) **Only ever fast-forward**

Fast-forward to the fetched branch. Fail if that is not possible. This is the standard behavior of `git pull`.

https://gitforwindows.org/

Best choice would be to use gits recommended option, currently **Fast-forward or merge**


Git 2.51.1 Setup

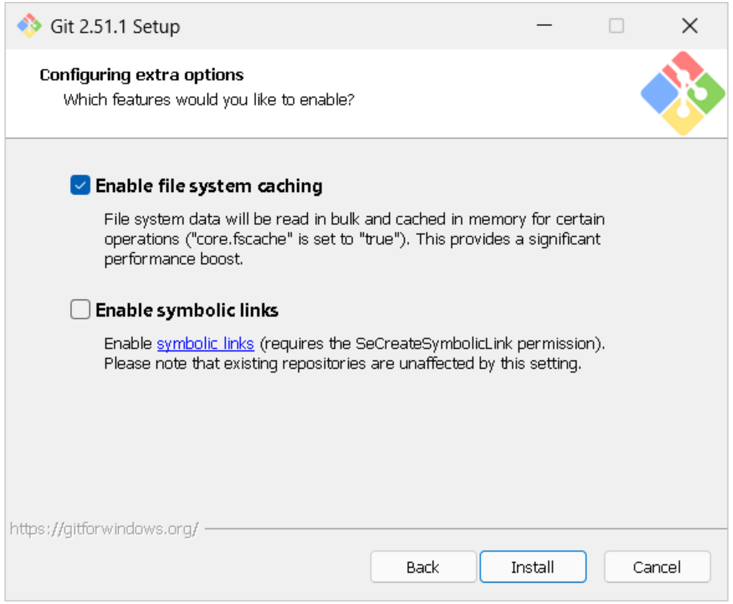
Choose a credential helper
Which credential helper should be configured?

[✓ Git Credential Manager]
Use the cross-platform Git Credential Manager.
See more information about the future of Git Credential Manager here.

None
Do not use a credential helper.

https://gitforwindows.org/

Windows allows you keep track of credentials like HTTPS based login on GitHub and Co.


Git 2.51.1 Setup

Configuring extra options
Which features would you like to enable?

[✓] Enable file system caching
File system data will be read in bulk and cached in memory for certain operations ("core.fscache" is set to "true"). This provides a significant performance boost.

[ ] Enable symbolic links
Enable symbolic links (requires the SeCreateSymbolicLink permission). Please note that existing repositories are unaffected by this setting.

https://gitforwindows.org/

Finally, **Install** `git` on your system

Install Git – MacOS

Homebrew

Install homebrew if you don’t already have it, then:

```
$ brew install git
```

Xcode Command Line Tools

Apple ships a binary package of Git with Xcode Command Line Tools. You can install this via:

```
$ xcode-select --install
```

Install Git – Linux / BSD

Debian/Ubuntu*/Mint

```
$ sudo apt install git
```

Fedora/Rocky/Alma

```
$ sudo dnf install git
```

Arch Linux

```
$ sudo pacman -S git
```

NixOS

```
$ sudo nix-env -it git
```

FreeBSD

```
# pkg install git
```

OpenBSD

```
# pkg_add git
```

** For Ubuntu, to get the latest stable Git version, use this PPA*

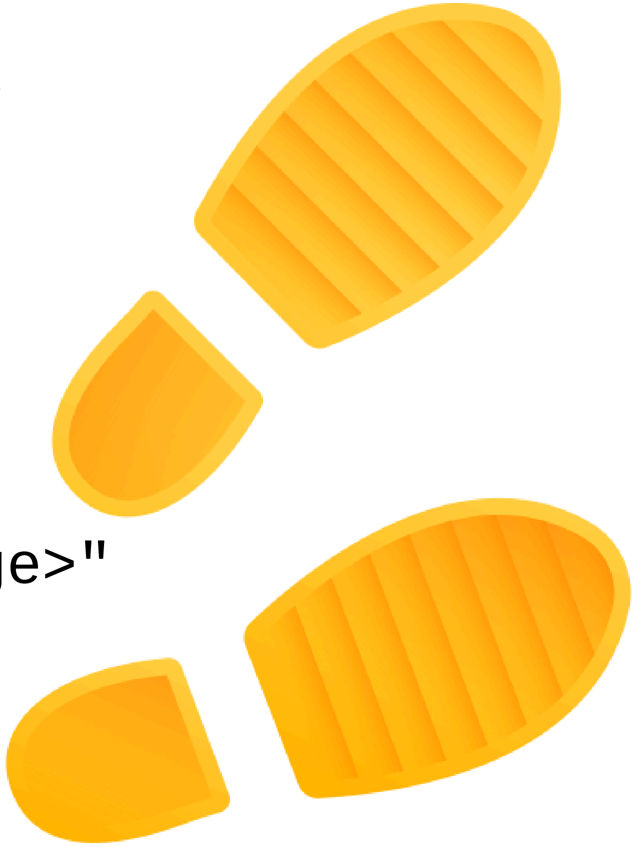
```
$ sudo apt-get-repository ppa:git-core/ppa
$ sudo apt update; sudo apt install git
```

First steps with Git

We will start with **local git operations** and **commands**

- `git init`

- `git add <filename>`
- `git add .`
- `git commit -m "<your git commit message>"`
- **`git status`**
- `git log`

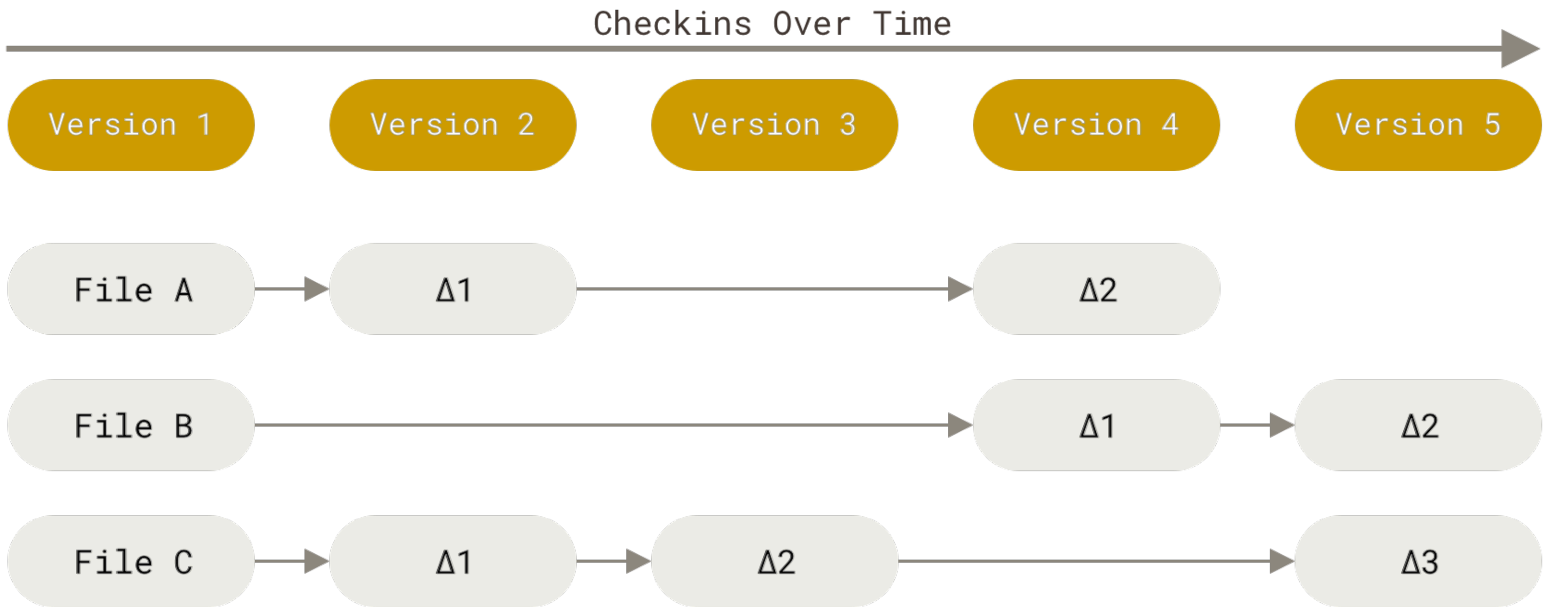


Git Repository

- The Git repository is represented by a directory named **`.git`** in your **project root directory**.
- With `git init`, a **new repository** will be created.
- All versions that are created with a `git commit` will be *stored* in there.
- Do **not** touch this directory without purpose!
- Each “project” will have it’s own git repository.

Major difference to other VCS...

Other VCS are storing versions with Delta-Information. Only the changes are stored and to get the current version all versions has to be combined.

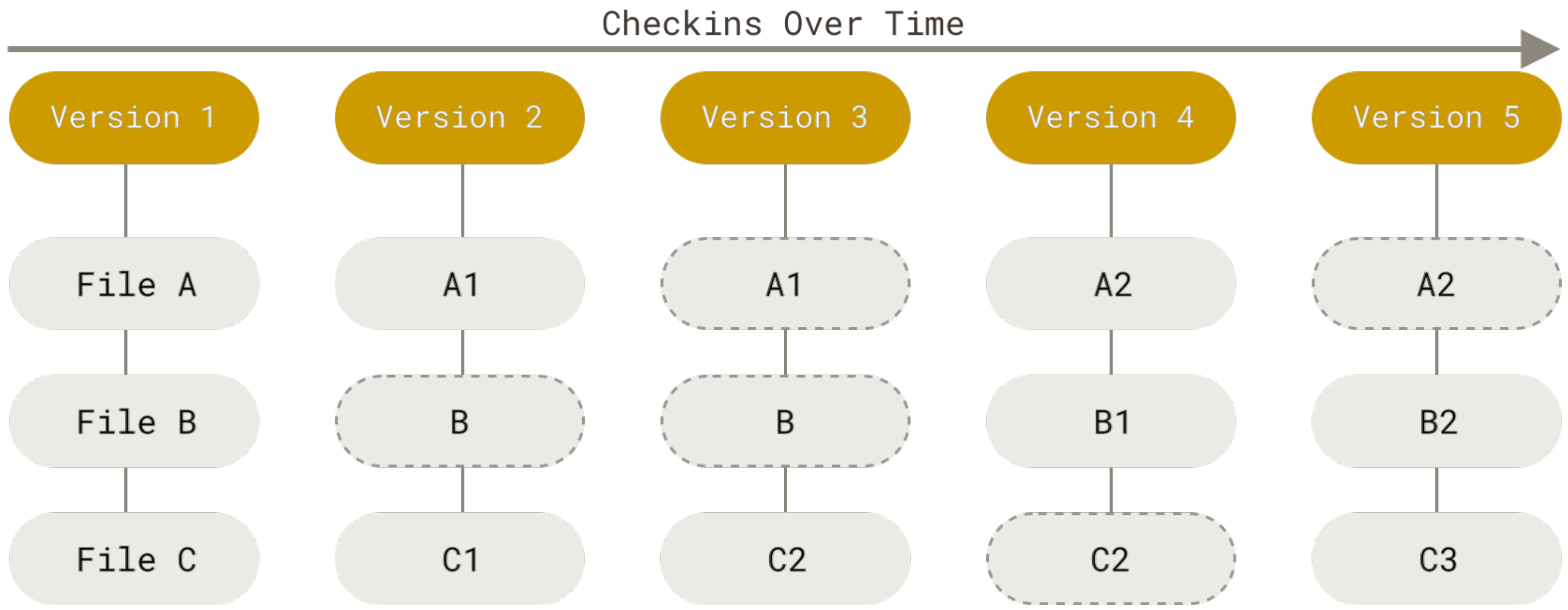


DIHSÜD Digital Innovation Hub

FH JOANNEUM University of Applied Sciences

… is the way of storing each version

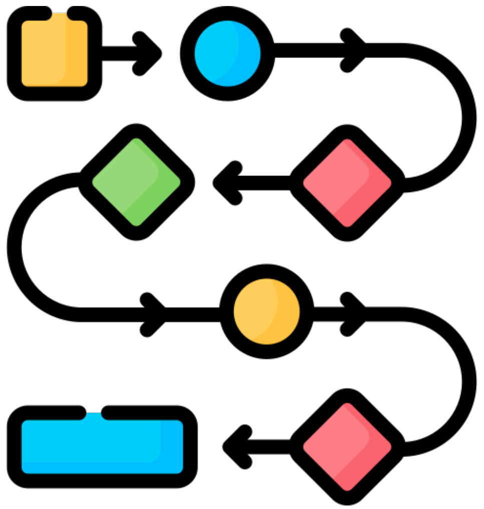
Git is storing **each new version of a file**, creates a new version “number” of all files, and refers to existing and not changed files.



Concept of local operations

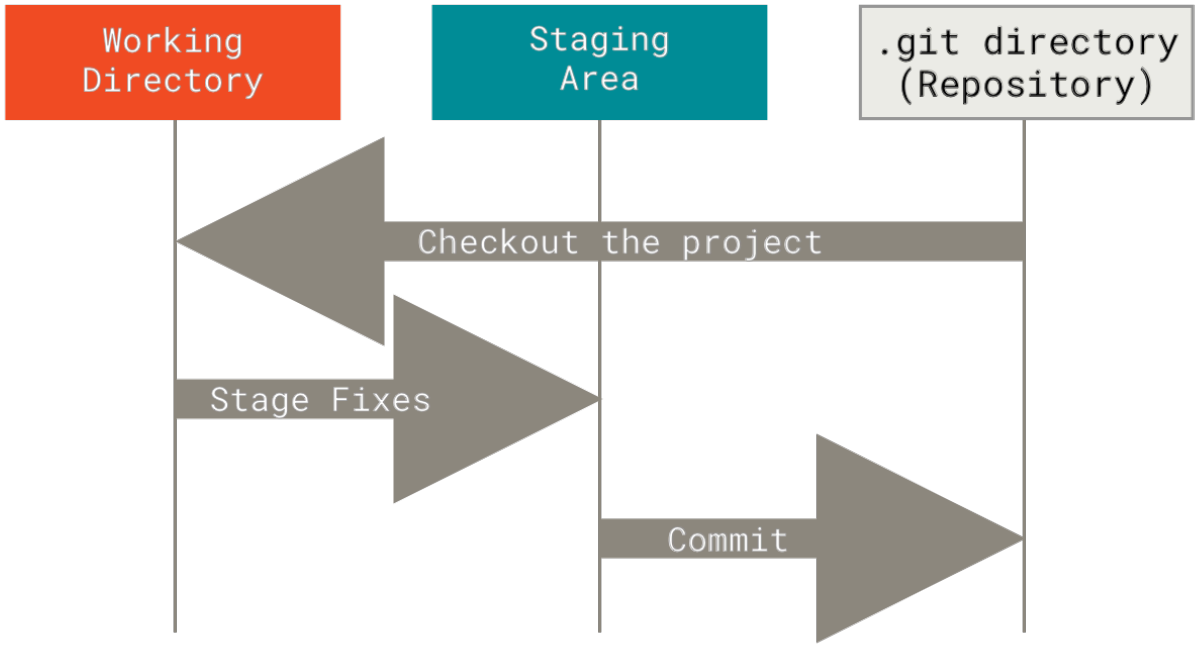
- **Most operations** with only local files and resources
- No information from another computer / server is needed
- **Offline working** nearly every time is possible
- Also **complete history is available**, because of complete clone

```
git status
git add
git commit
git log
git diff
```



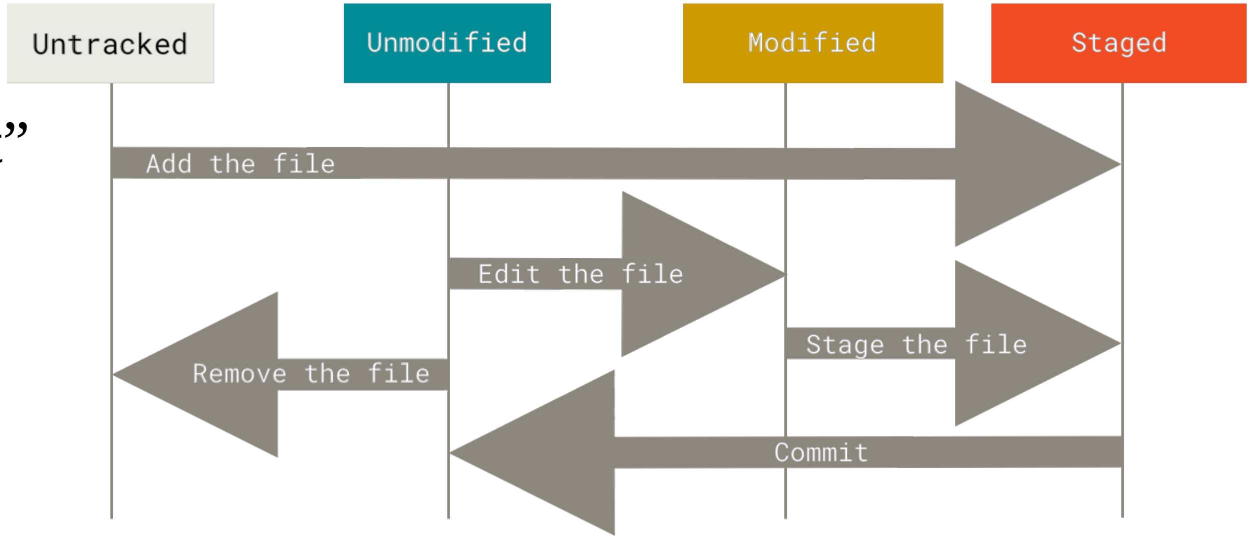
Git Workflow *(local)*

1. Making changes in **working directory**
   - add new file
   - remove file
   - modify file
2. Add changes to **staging area**
   - `git add <fileA> <fileB> ...`
3. **Commit** changes to create a new version
   - `git commit -m "<commit message>"`



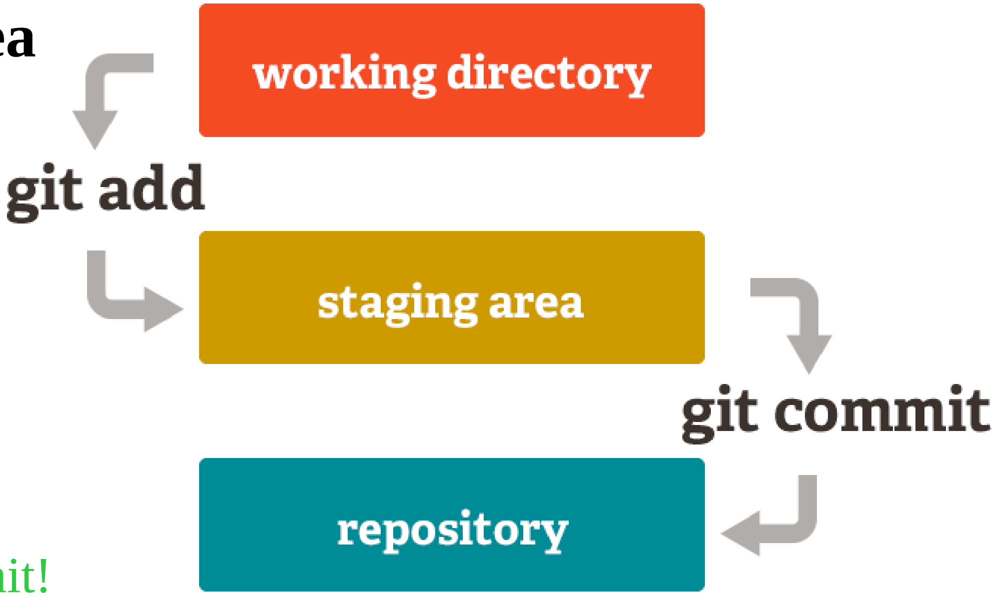
Recording Changes to the Repository

- New files are **untracked**
- **Tracked** files are already versioned
- Changes to files leads to “**modified**”
- Modified files become “**staged**”
- A group of staged files become *unmodified* state through a “**commit**”



Stages of a file in Git

1. **Modified**
   - Editing files
2. **Staged**
   - Finished editing, transferred to **staging area** and ready to create a new version
3. **Committed** (*unmodified*)
   - The **repository** has a new version
4. *Untracked*
   - The repository doesn't know the file



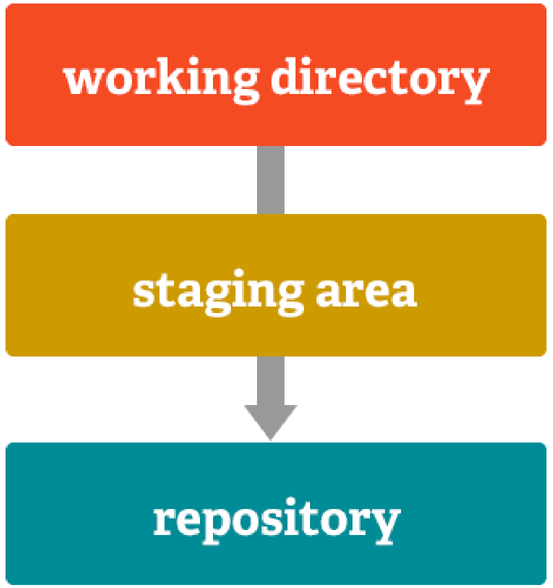
You can stage and commit multiple files by a single commit!

https://git-scm.com/about/staging-area

Quick way to commit all your changes

- Possible with one commands
- **Not recommended** since you loose control
- **Only commit working version!**

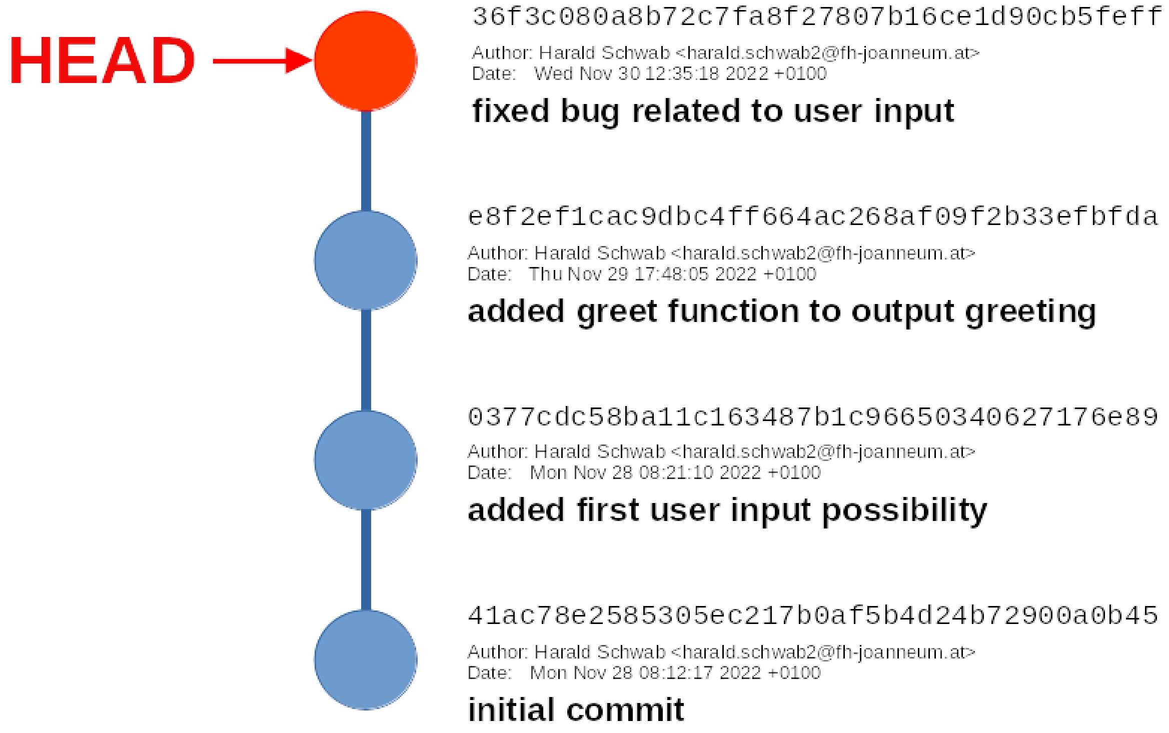
Why could this lead to problems?



git commit -a

Commit

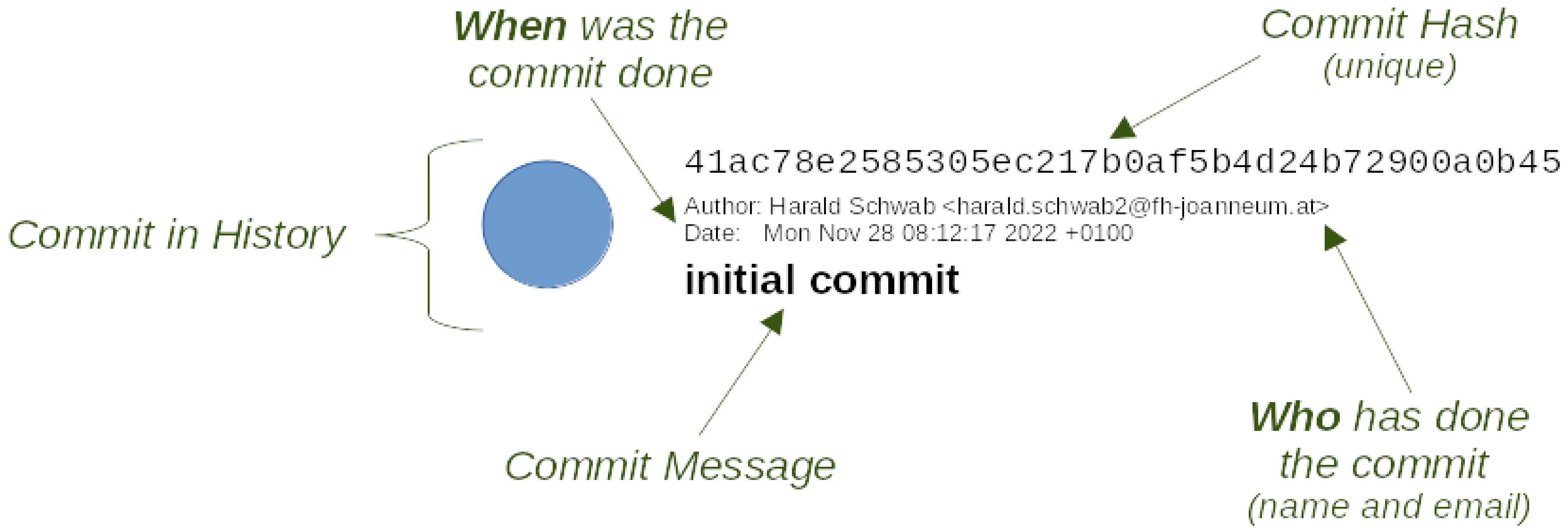
- Each commit has its own – **unique** – “id” (=hash)
- **HEAD** refers to the current *active* commit
- Components of a commit:
  - WHO: Author (name + email)
  - WHEN: Date + Time
  - WHAT: commit message



```
36f3c080a8b72c7fa8f27807b16ce1d90cb5feff
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Wed Nov 30 12:35:18 2022 +0100
fixed bug related to user input

e8f2ef1cac9dbc4ff664ac268af09f2b33efbfda
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Thu Nov 29 17:48:05 2022 +0100
added greet function to output greeting

0377cdc58ba11c163487b1c96650340627176e89
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Mon Nov 28 08:21:10 2022 +0100
added first user input possibility

41ac78e2585305ec217b0af5b4d24b72900a0b45
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Mon Nov 28 08:12:17 2022 +0100
initial commit
```

https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History

Components of a commit



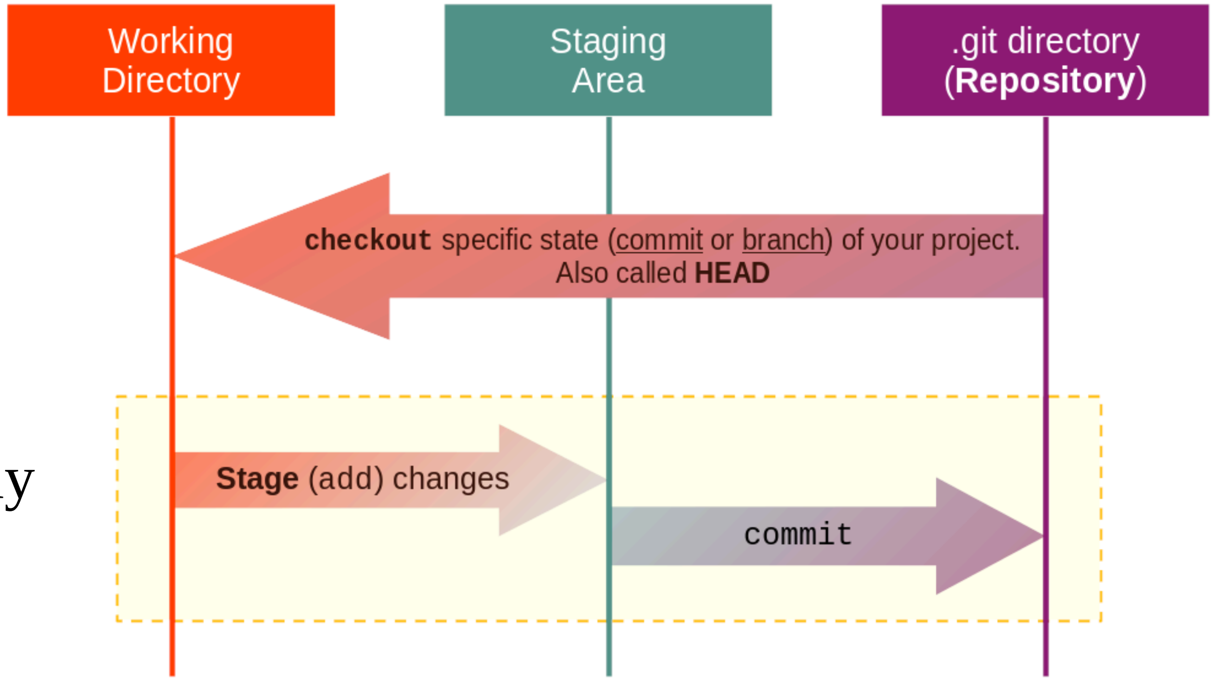
When was the commit done
Commit Hash (unique)
Commit in History
Commit Message
Who has done the commit (name and email)

```
41ac78e2585305ec217b0af5b4d24b72900a0b45
Author: Harald Schwab <harald.schwab2@fh-joanneum.at>
Date:   Mon Nov 28 08:12:17 2022 +0100
```

initial commit

Basic Git workflow for every developer

1. **Modify/Create** files
   - in the **working directory**[3]
2. **Stage** (`git add`) files
   - to the **staging area**
3. **Commit** changes (`git commit`)
   - and store a snapshot permanently in the **repository**


Working Directory
Staging Area
.git directory (Repository)
checkout specific state (commit or branch) of your project. Also called HEAD
Stage (add) changes
commit

[3] *When talking about working directory, the local git repository is meant. The directory where the `.git` folder is located. Typically your project root directory.*

Git commands – Configuration and Initialisation

- `git config`: configure git[4]
  - `git config --global user.name`: check/set the name that should be used for your commits. Has to be set **once**!
  - `git config --global user.email`: check/set the email that should be used for your commits. Has to be set **once**!
    - `git config --local`: use `--local` instead of `--global` to set specific configurations individual for the current local repository
  - `git config --global init.defaultbranch`: What should be the name of the default branch? *(today, typically main)*
- `git init`: initialise a new git repository in the current directory. Only needed **once** per project at the beginning. Will create the `.git`-directory.

[4]Global configurations could be found under `~/.gitconfig`

Git commands – Staging

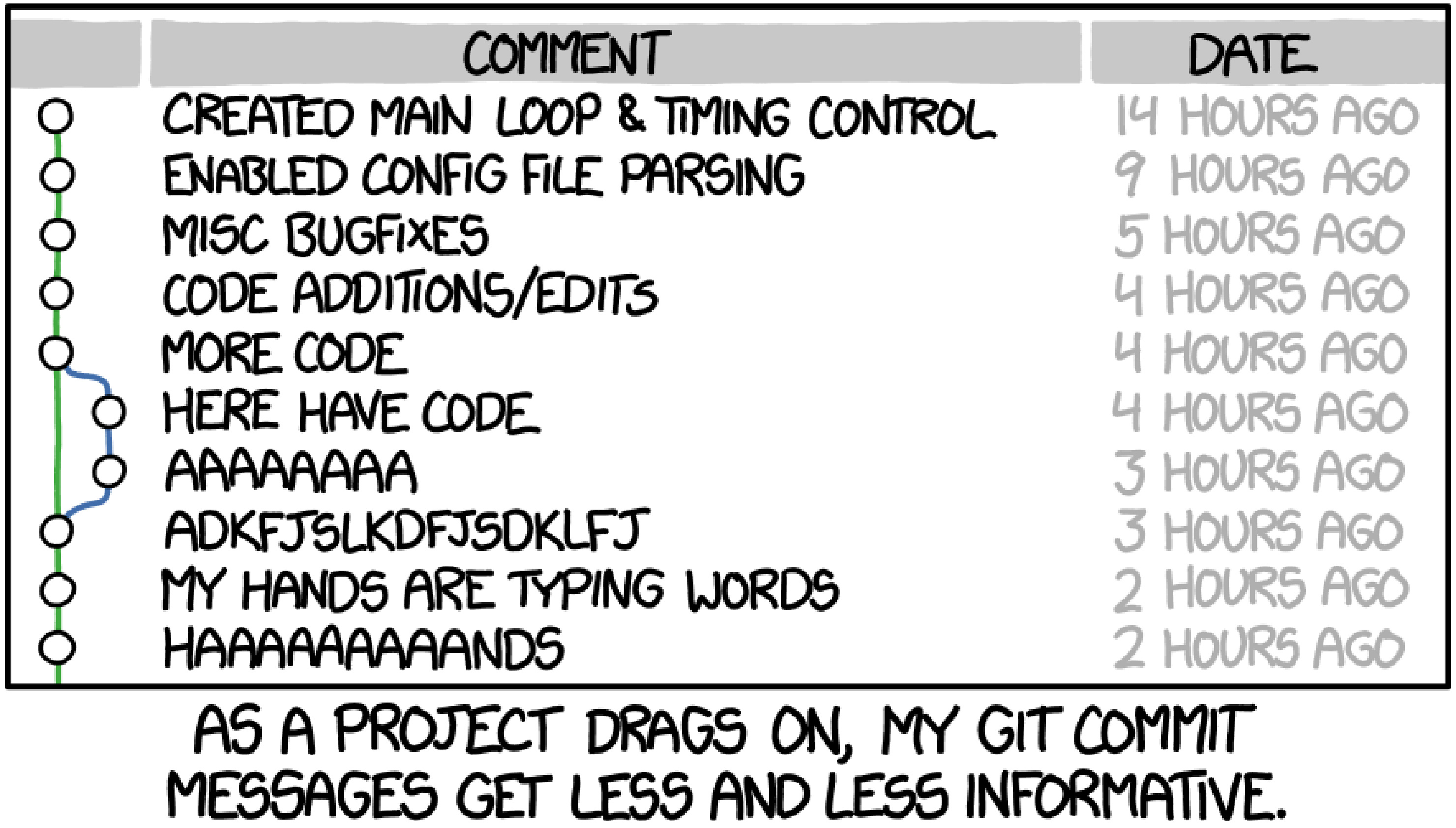
- `git status`: show current state of working directory and staging area
  - `git status -s`: will print a compacter output
- `git add`: add new/changed files to the staging area
  - `git add .` or `git add *` will add all files at once. Use `git add <path/to/file>` to add a specific file
- `git diff`: Show difference between the working directory and the staging area
- `git reset`: remove files from staging
  - `git reset HEAD` will remove all already staged files from staging area. *It will **not** affect any changes since the last commit in the files itself.*
  - `git reset <filename>` will remove the file from staging. *It will **not** affect any changes in the file itself.*
  - `git reset <commit-hash>` will **revert** all changes until this commit.

Git commands – Commit and Git-History

- `git commit`: commit all staged changes (will open the set *default editor*)
  - `git commit -m "<commit message>"` allows to provide the message directly
  - `git commit -am "<message>"`: `-a` will add all changed (*not untracked*) files automatically to this commit
- `git log`: show the history of your (*local*) repository
  - `git log --oneline` provides a more compact output
- `git checkout`: switch to a specific commit/reset all uncommitted changes of a file
  - `git checkout <filename>` will **revert** all changes in this files since last commit! *Use it with care, this operation could not be undone!*
  - `git checkout <commit-hash>` will switch[5] to the specific commit. You can switch back to the "latest" commit with `git checkout main` *(as long your branch is called main)*

[5] old way to switch branches. Use `git switch <branch>` instead

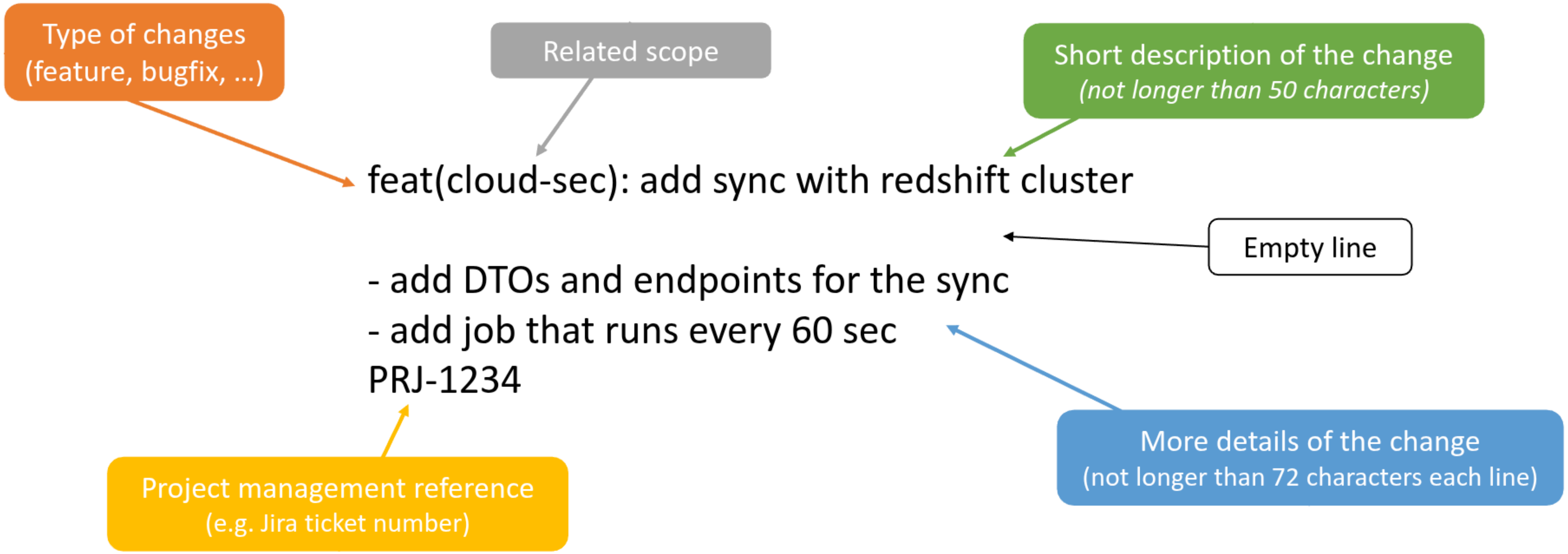
Commit messages, … which you should not write!



	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.

Meaningful commit messages


Type of changes (feature, bugfix, …)
Related scope
Short description of the change (not longer than 50 characters)
Empty line
More details of the change (not longer than 72 characters each line)
Project management reference (e.g. Jira ticket number)

```
feat(cloud-sec): add sync with redshift cluster

- add DTOs and endpoints for the sync
- add job that runs every 60 sec
PRJ-1234
```

See also Conventional Commits

https://www.freecodecamp.org/news/how-to-write-better-git-commit-messages/

Not every file has to be tracked

- Not every kind of file should be part of a repository
- Let Git **ignore** files that you don’t want to track
  - create a **.gitignore** file
  - will include a list of files and directories, which will **NOT** be tracked anymore
- Files to ignore, e.g.:

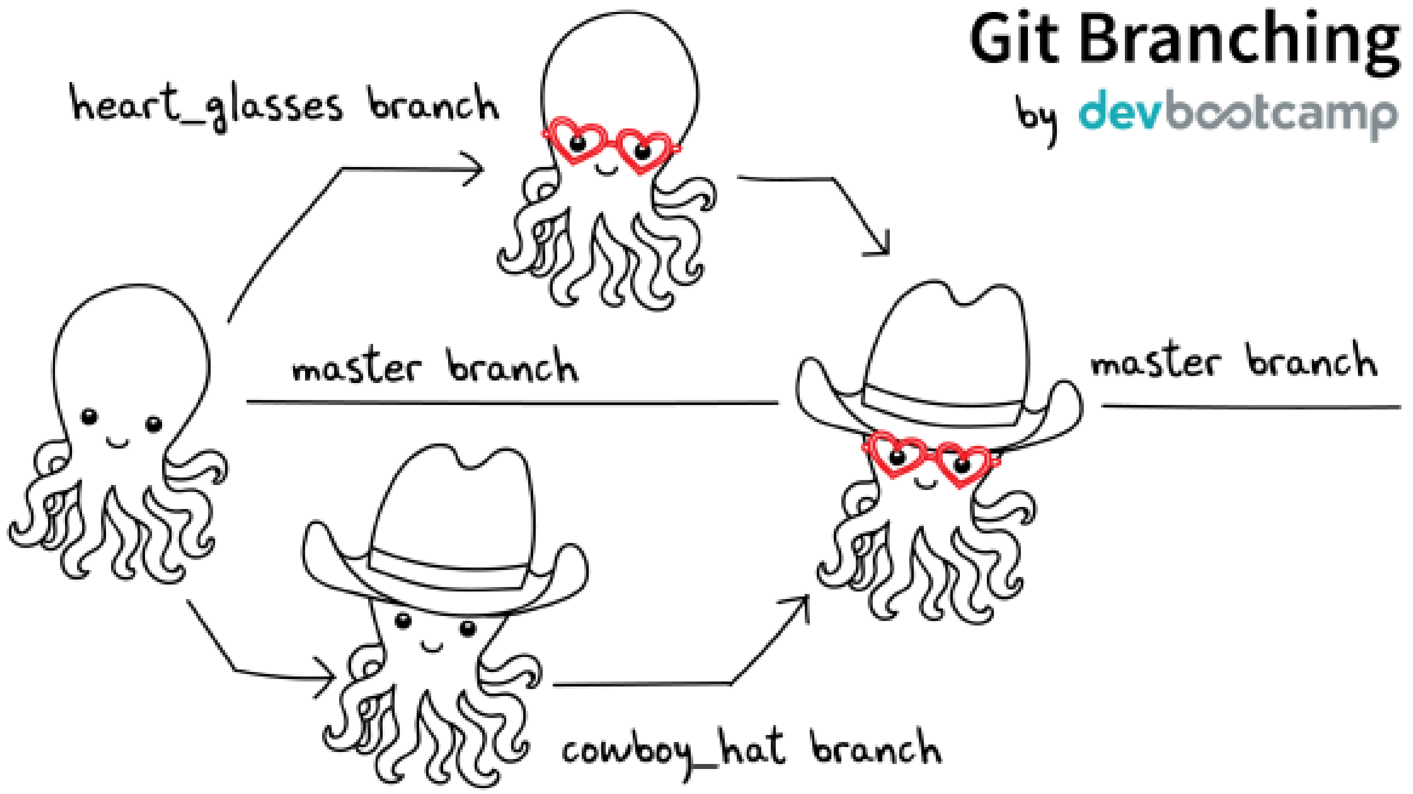
▸ Executables	*.exe
▸ Generated files	*.class
▸ Images	*.iso, *.dmg
▸ Log files	*.log

  - 3rd party libraries (use a package manager)
  - Secrets (passwords, logins, api keys, …)
  - Backup files *(automatically)* created by your editor/ide
  - Other example https://gist.github.com/octocat/9257657

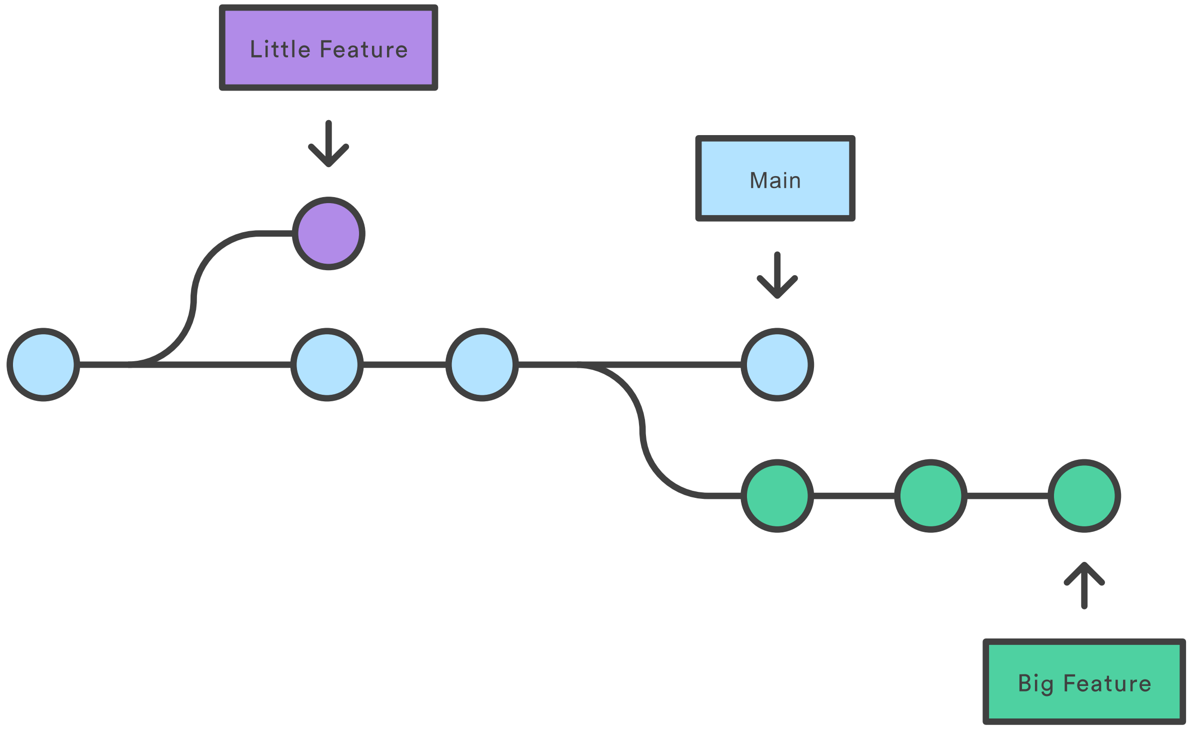
NEVER share secrets in a git repository!

Branches

Git Branches



Git Branches



- Branches are **lightweight** and **heavily used** in daily development activities
- A branch is a **reference to a commit** (nothing is copied)
  - Branch is tip of a series of commits
- At least one branch (typically "**main**") exists in every repository

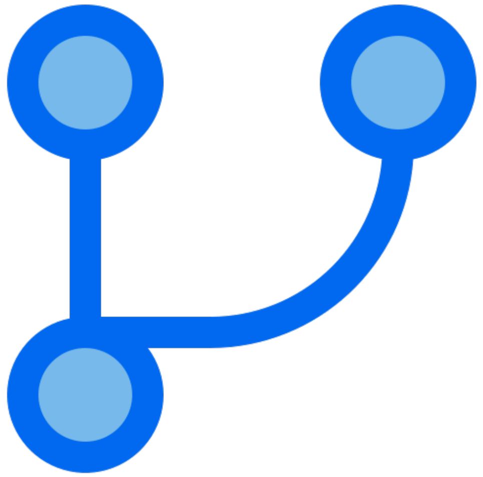
https://www.atlassian.com/git/tutorials/using-branches

A new branch…

… illustrates an independent line of development.

We create a new branch when
- **Testing** first ideas for development
- **Work** on feature(s) or any other issues
- **Fixing** bugs

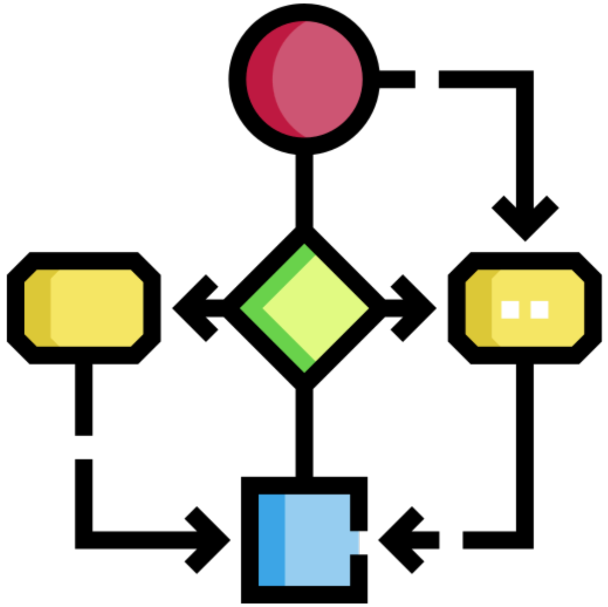
Think of it like an independent brand-new:
- working directory
- staging area
- project history



Branches Workflow

Working with branches, we are able to:

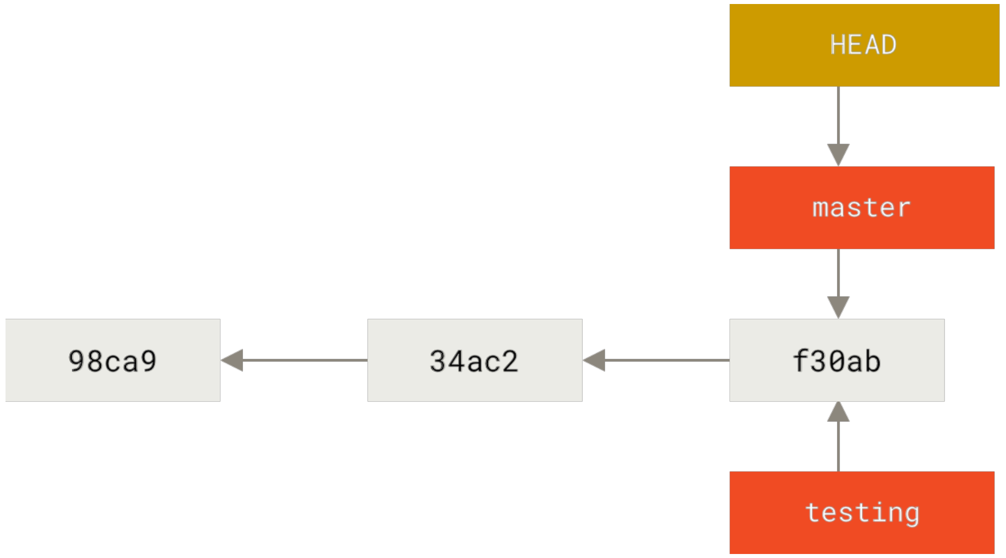
- **Choose** in which branch we want to work at the moment → `git switch`
- **Create** new branches → `git branch`
- **Merge** multiple branches together → `git merge`



Creating a new branch

```
git branch <branch-name>
```

Creates a new branch, pointing on the current active commit (*HEAD*)



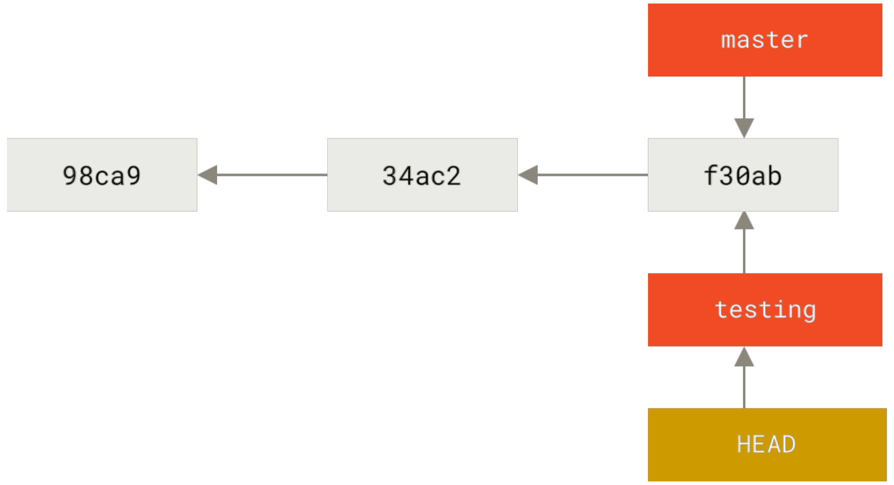
```
git branch testing
```

https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell#_create_new_branch

Switch branches

`git switch <branch-name>`[6]

Switch branch means to move *HEAD* to commit where the branch is currently pointing to

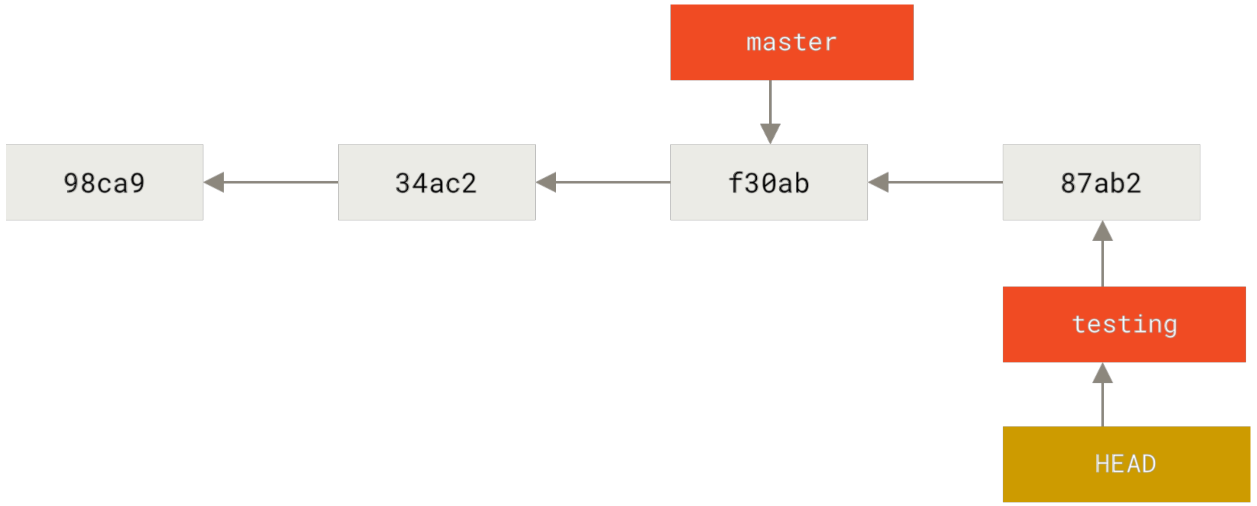


```
git switch testing
```

[6] `git checkout <branch-name>` will also work and is the *old* way of switching branches

Commit on *new* branch

Making changes and a **`commit`** on the new branch will create a new version. The other branch still points to the *old* commit.

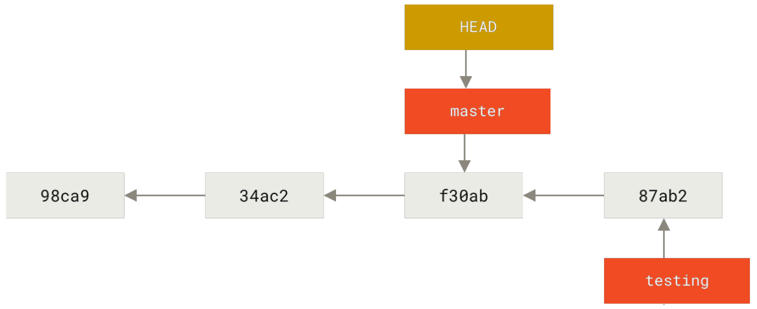


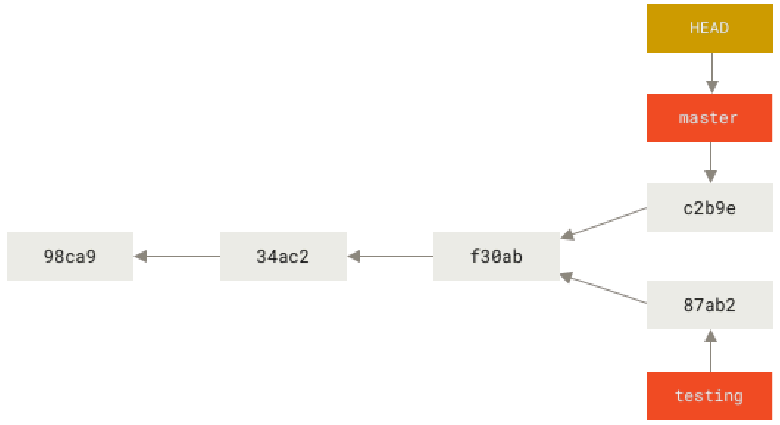
```
vim main.py
git commit -a -m \
  'Make some change in testing'
```

Advance other branch

```
git switch master
```

Moving back to *other* branch





Doing some changes and make new commit

```
vim main.py
git commit -a -m \
  'Make other changes in master'
```

Git Branching

- `git branch`: list all available branches
- `git branch <name>`: create new branch with *name*
  - `git branch -d <name>`: delete branch with *name*
- `git switch <name>`: switch to branch with *name*
  - `git switch -c <name>`: **create** and switch to branch
- *`git checkout <name>`: switch to branch*[7]

```
$  git branch
* testing
  main
```

Active branch → *testing*

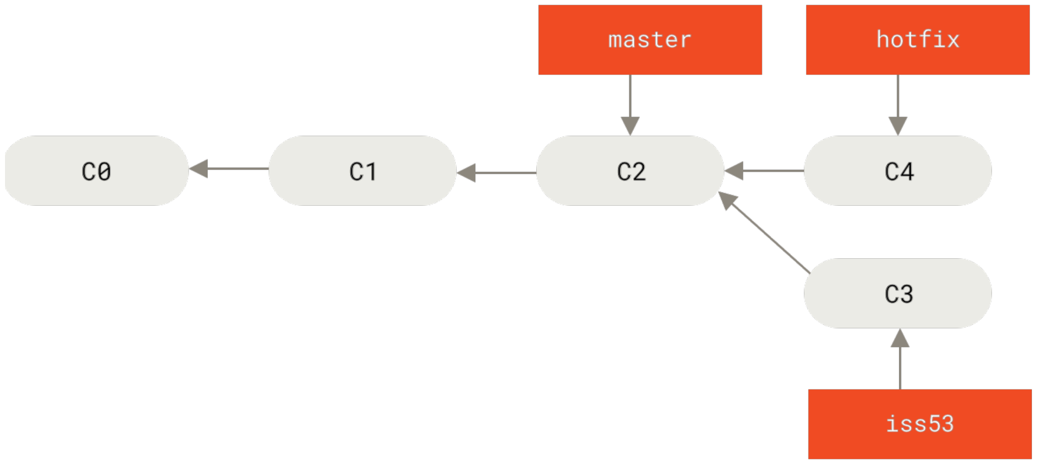
```
$ git log --oneline
193fed9 (HEAD -> testing) add some more comments
f6c4cc2 add documentation
66599c7 add two new examples
eb85d9a (main) fix: format numbers to be sorted
8c88734 feat: README added with example usage
2f3584b fix: do not overwrite, use a counter instead
```

With `git log`, branches pointing on commits will be shown. `HEAD -> testing` means that this is the current active commit/version.

[7] *`git switch` was added in Version 2.23 of Git. Before, `git checkout` was used to switch branches.*

Git Merging

How can we **merge** changes of different branches back together?

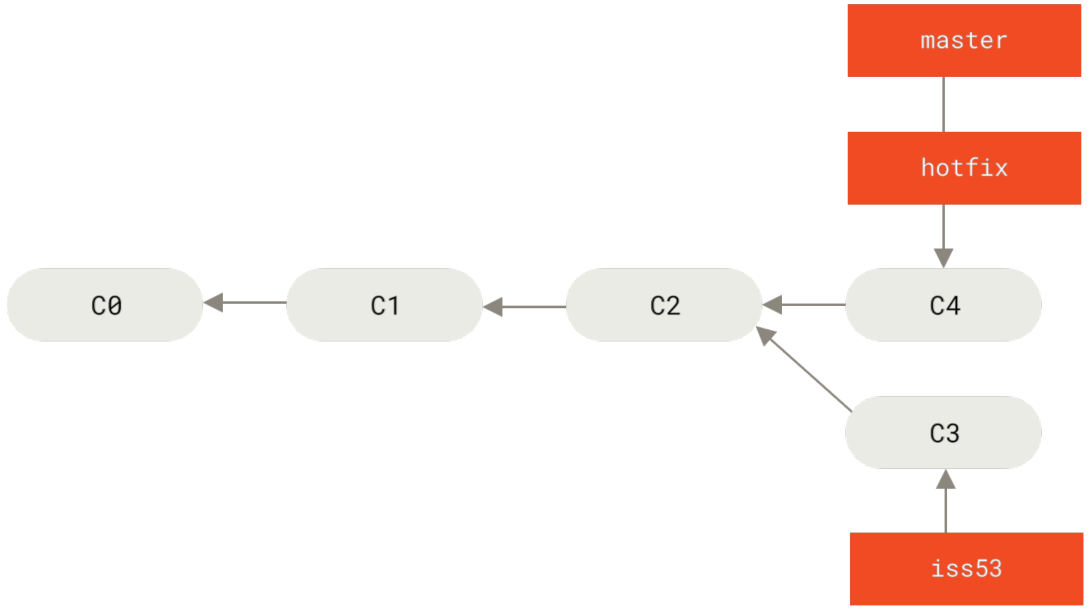


1. switch to target branch *(where changes should be merged to)*
2. **`git merge`** `<source-branch>`

```
git switch master
git merge hotfix
```

We want to merge the changes from *hotfix* back to *master*

Removing *unneeded* Branch

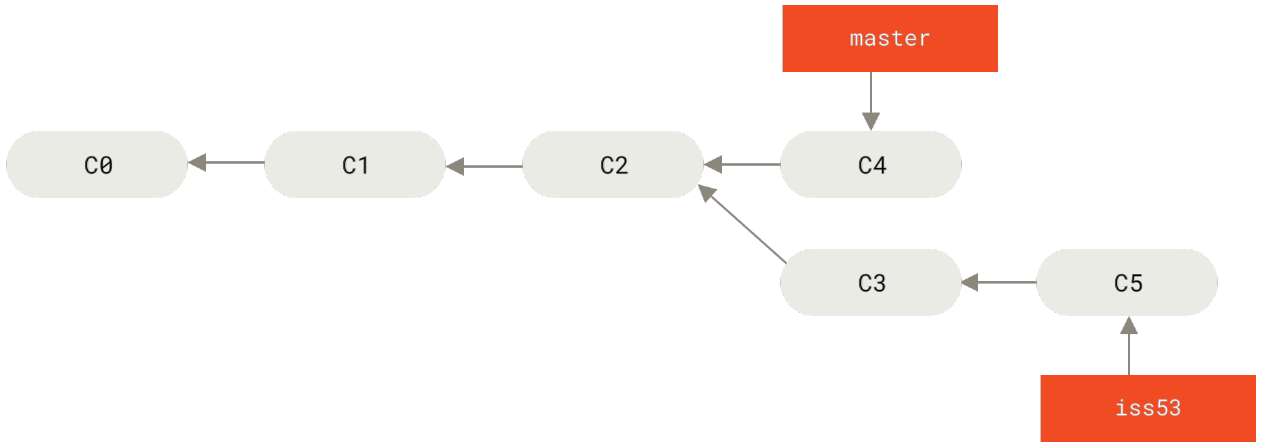


```
git branch -d hotfix
```

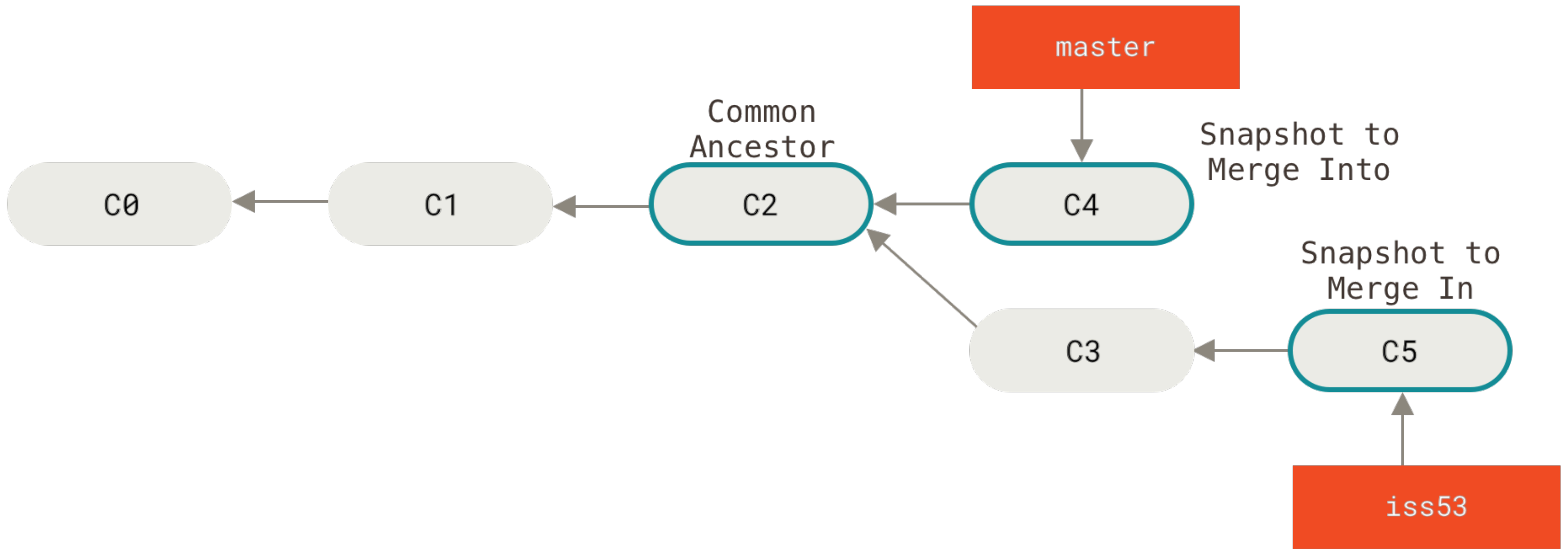
After we merged our *hotfix* branch with *master* we could remove the unneeded branch.

Merging *separated* branches

As long as there are no different versions between branches, the pointer is simply moved. But what if the branches have advanced separately?

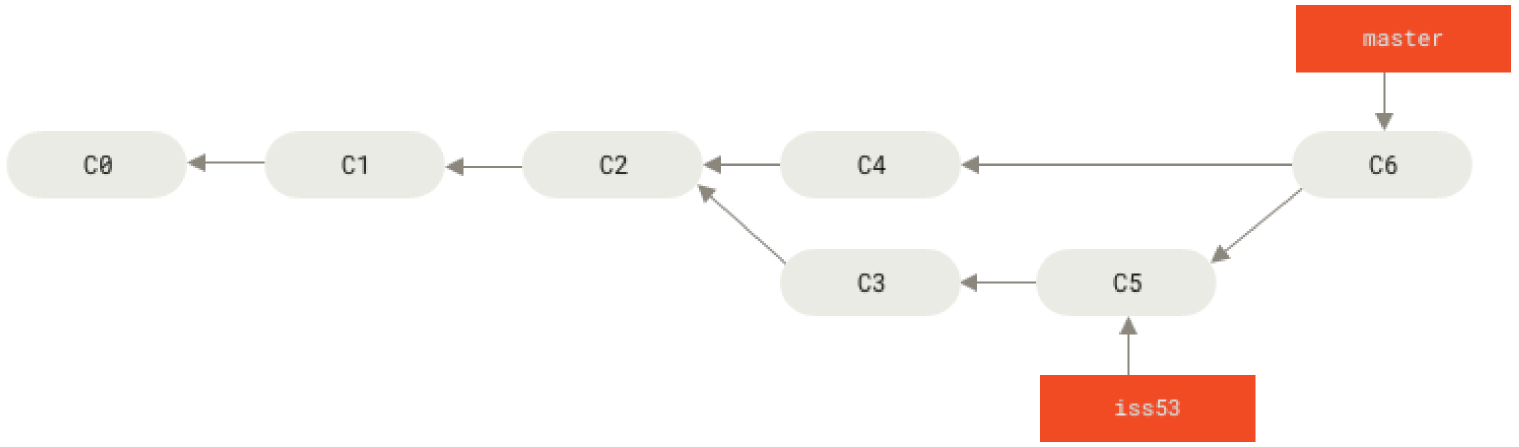


Basic merging



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

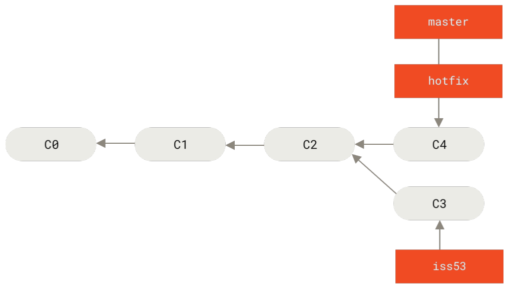
Basic merging



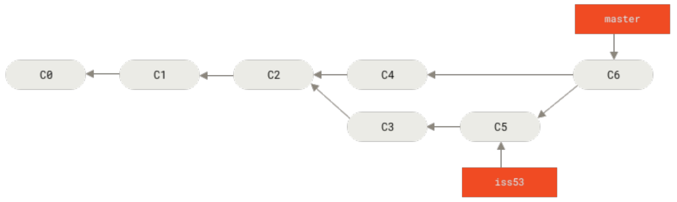
Basic merging

Merge: Changes from branch A are applied to branch B.

- If the origin is in a line *(only one of the two branches contains new versions)*, the pointer is simply moved.

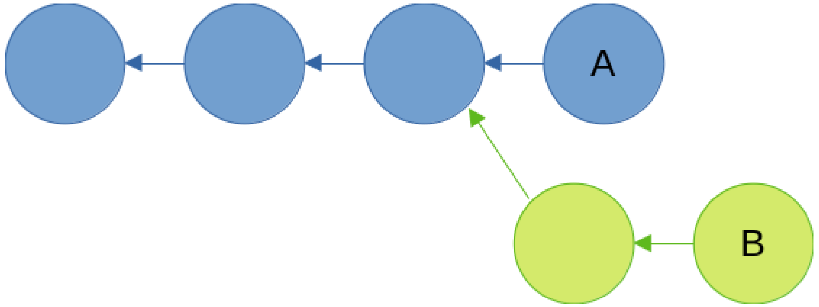


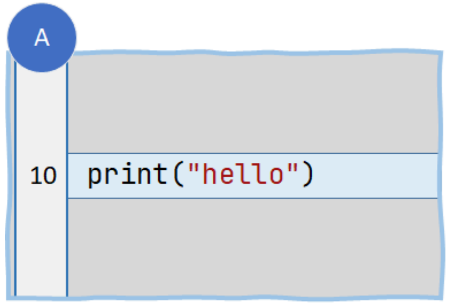
- Once both branches have been advanced, a new version is always created from the merge.



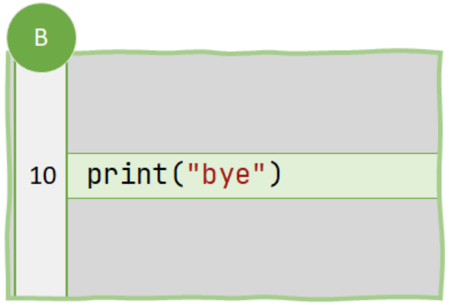
How merge works

How does git know how merges are to resolve?





```
10  print("hello")
```

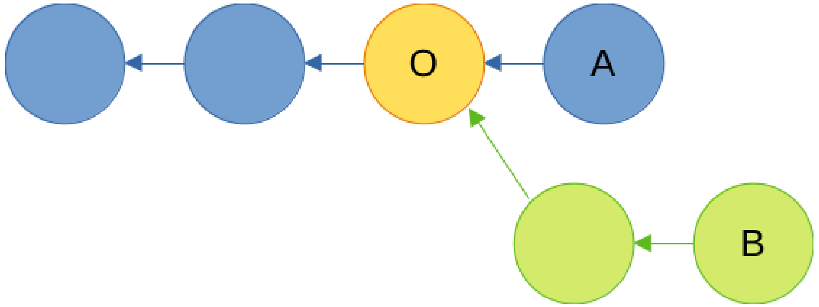


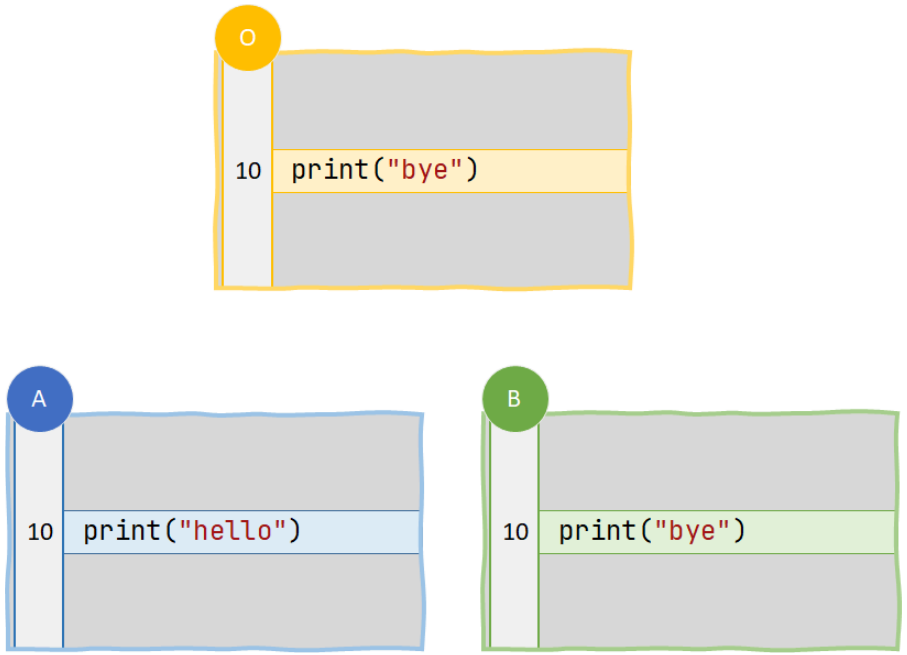
```
10  print("bye")
```

2-way merge. How to decide which version is to choose?

3-way merge

Solution: 3-way merge





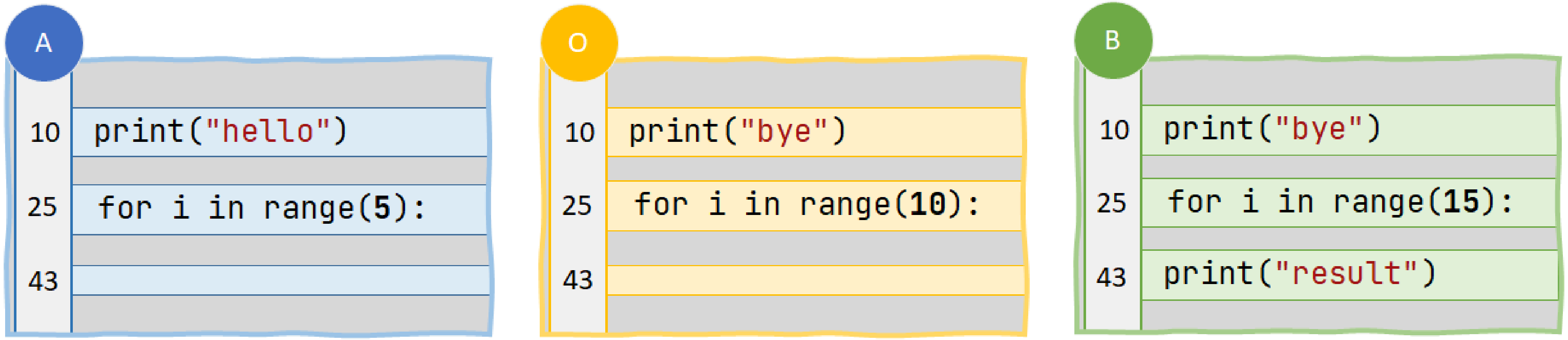
```
10  print("bye")
```

```
10  print("hello")
```

```
10  print("bye")
```

Git compares branch A with B **and** the origin.

Automatic Merge



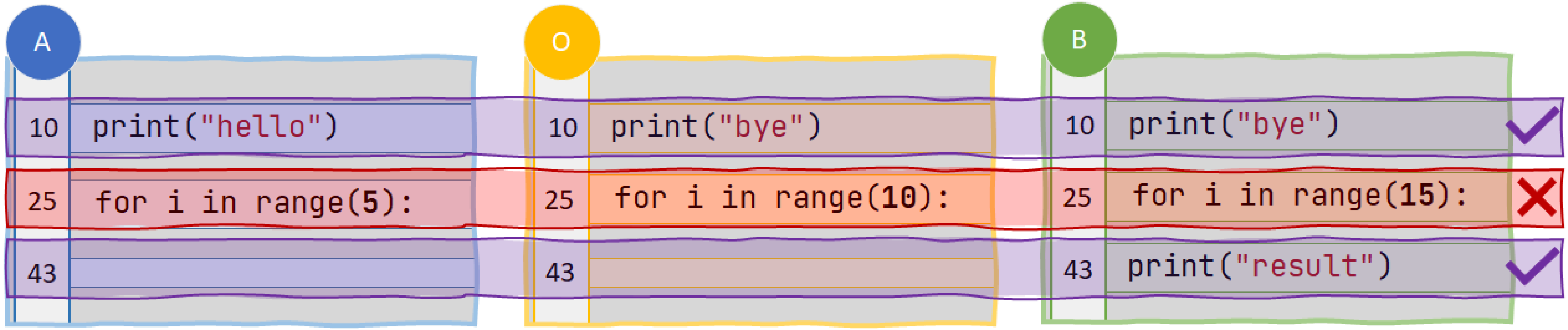
```
print("hello")

for i in range(5):

```

```
print("bye")

for i in range(10):

```

```
print("bye")

for i in range(15):

print("result")
```

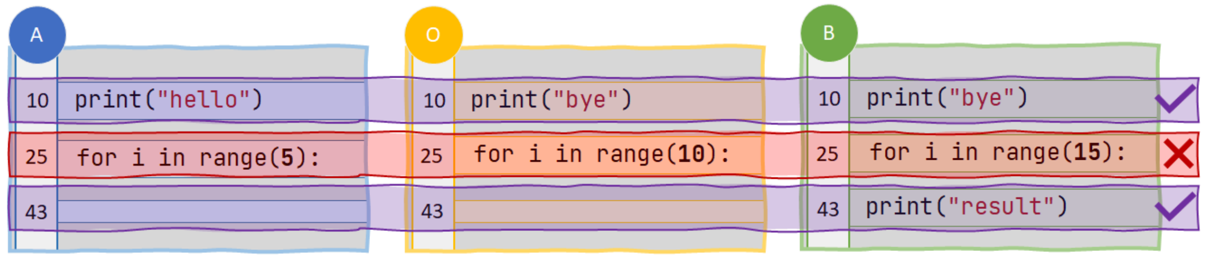
Which version is to keep?

Merge conflict



- Lines 10 and 43 can be merged automatically by Git
- But what’s about line 25?

Conflict: manual resolve necessary



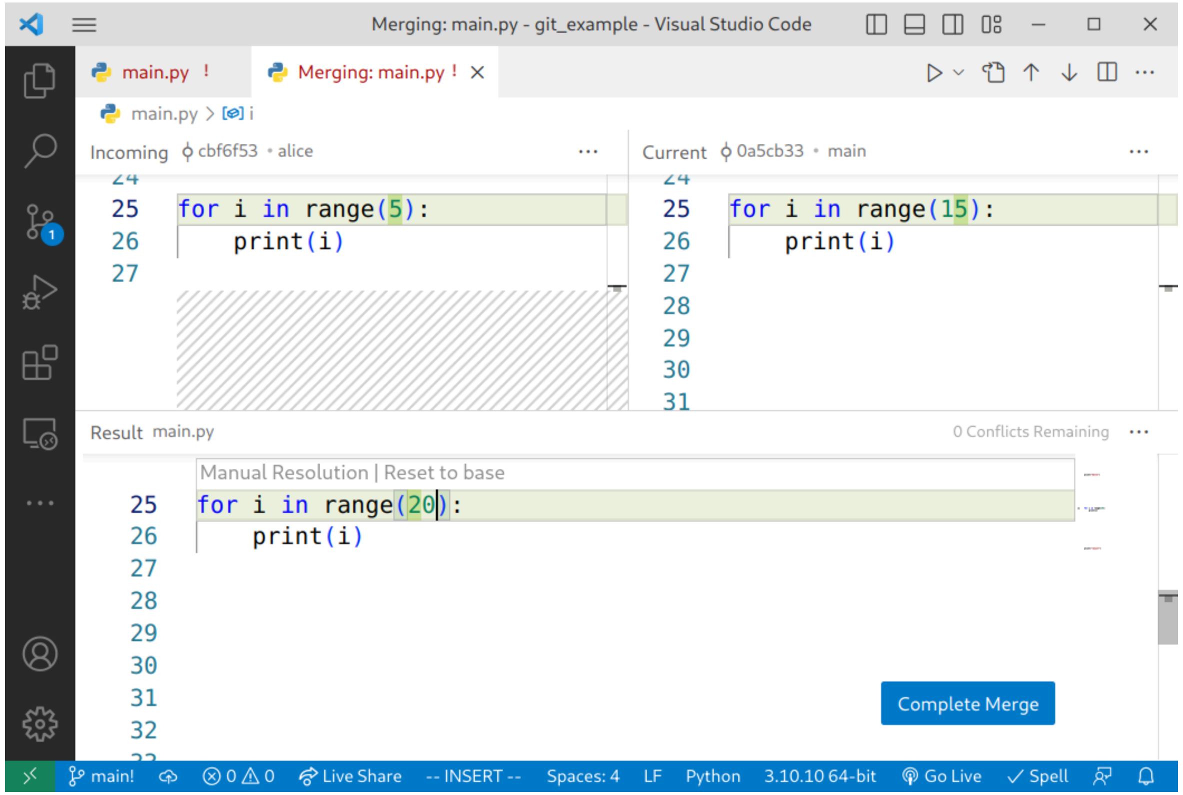
```
<<<<<<<< HEAD
for i in range(15)
========
for i in range(5)
>>>>>>>> featA
```

```
$ git merge featA
```

```
Auto-merging main.py
CONFLICT (content): Merge conflict in main.py
Automatic merge failed; fix conflicts and then
commit the result.
```

- Choose the correct solution between “conflict dividers”
- Add and commit the changes to resolve the conflict (and end the merge process)

Resolve merge conflicts with tools



Merging: main.py - git_example - Visual Studio Code
main.py !
Merging: main.py !
main.py > i
Incoming cbf6f53 • alice
Current 0a5cb33 • main
Result main.py
0 Conflicts Remaining
Manual Resolution | Reset to base
Complete Merge
main! 0 0 Live Share -- INSERT -- Spaces: 4 LF Python 3.10.10 64-bit Go Live Spell

Incoming:

```
for i in range(5):
    print(i)
```

Current:

```
for i in range(15):
    print(i)
```

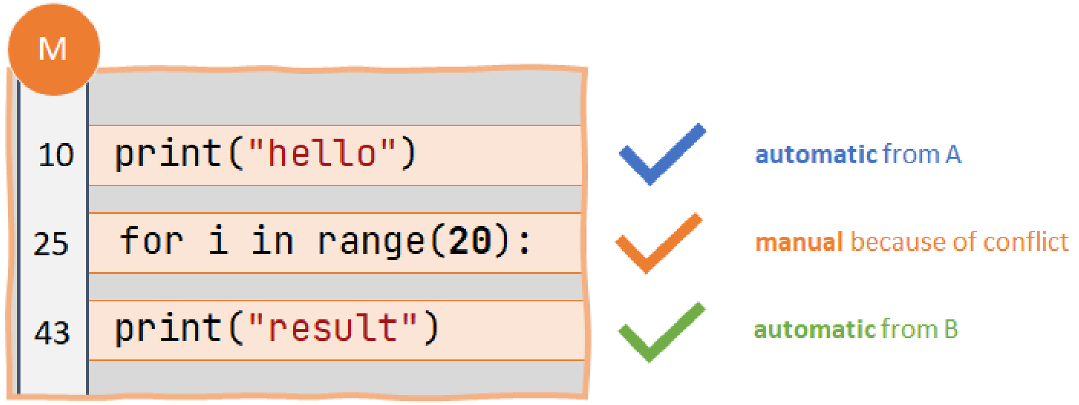
Result:

```
for i in range(20):
    print(i)
```

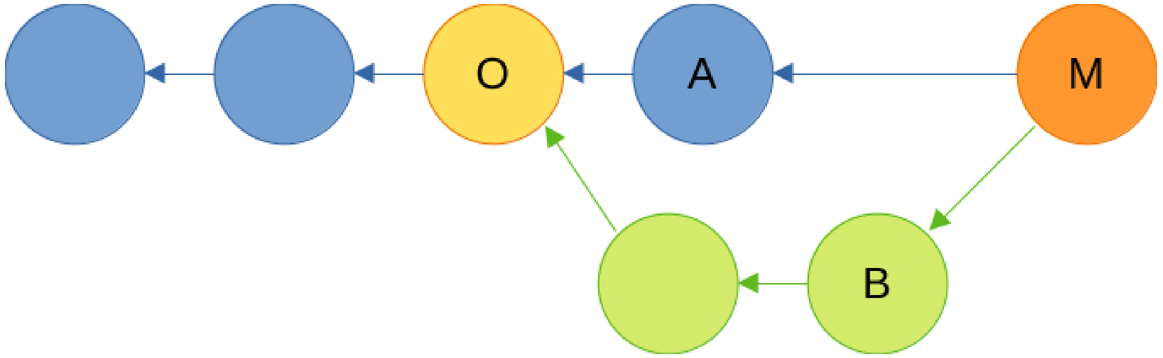
Graphical tools tend to be more effective at resolving conflicts due to their extended user interface.

For example:
VS Code with Merge Overview

Commit resolved merge *conflict*



```
10  print("hello")
25  for i in range(20):
43  print("result")
```

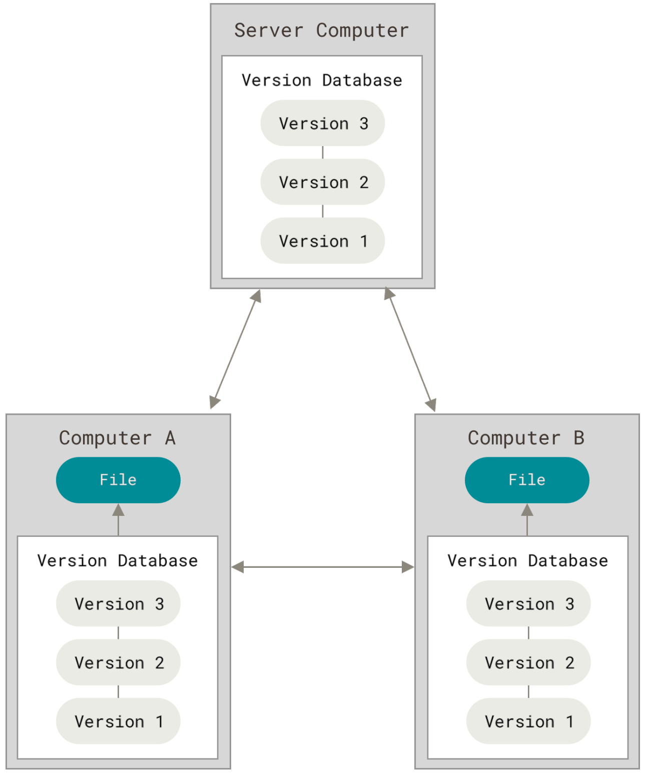


- After you resolved conflicts manually, create a new version with a commit

```
git add .
git commit -m \
 "merged changes from branchB"
```

Remote

Working Collaboratively

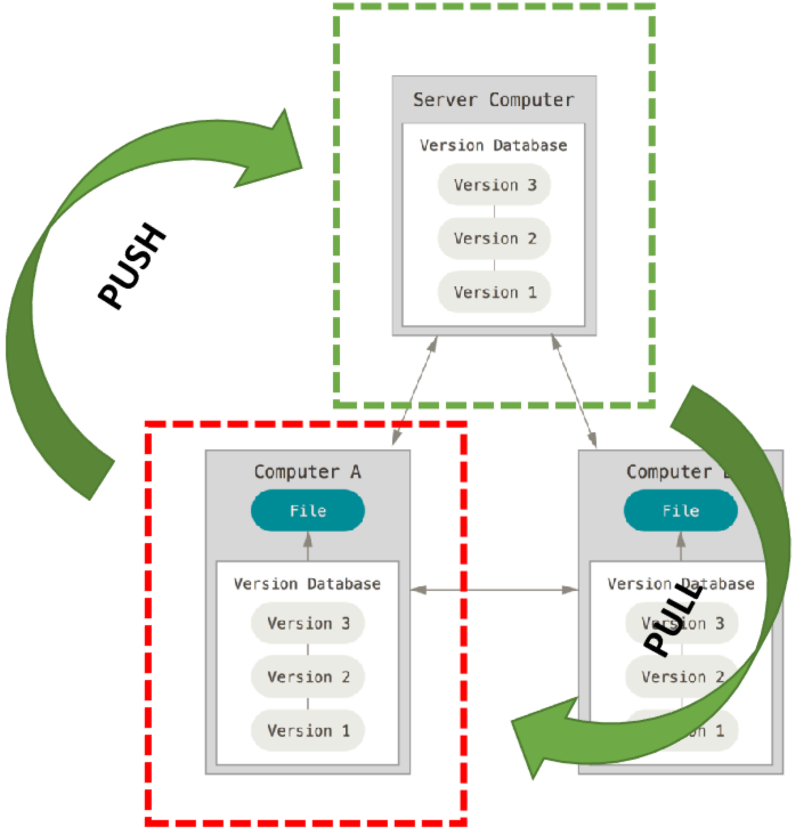


- Git is a **distributed** VCS
- Simply share your code base with other
- Add **remote** repositories
- **push** your changes or **pull** others

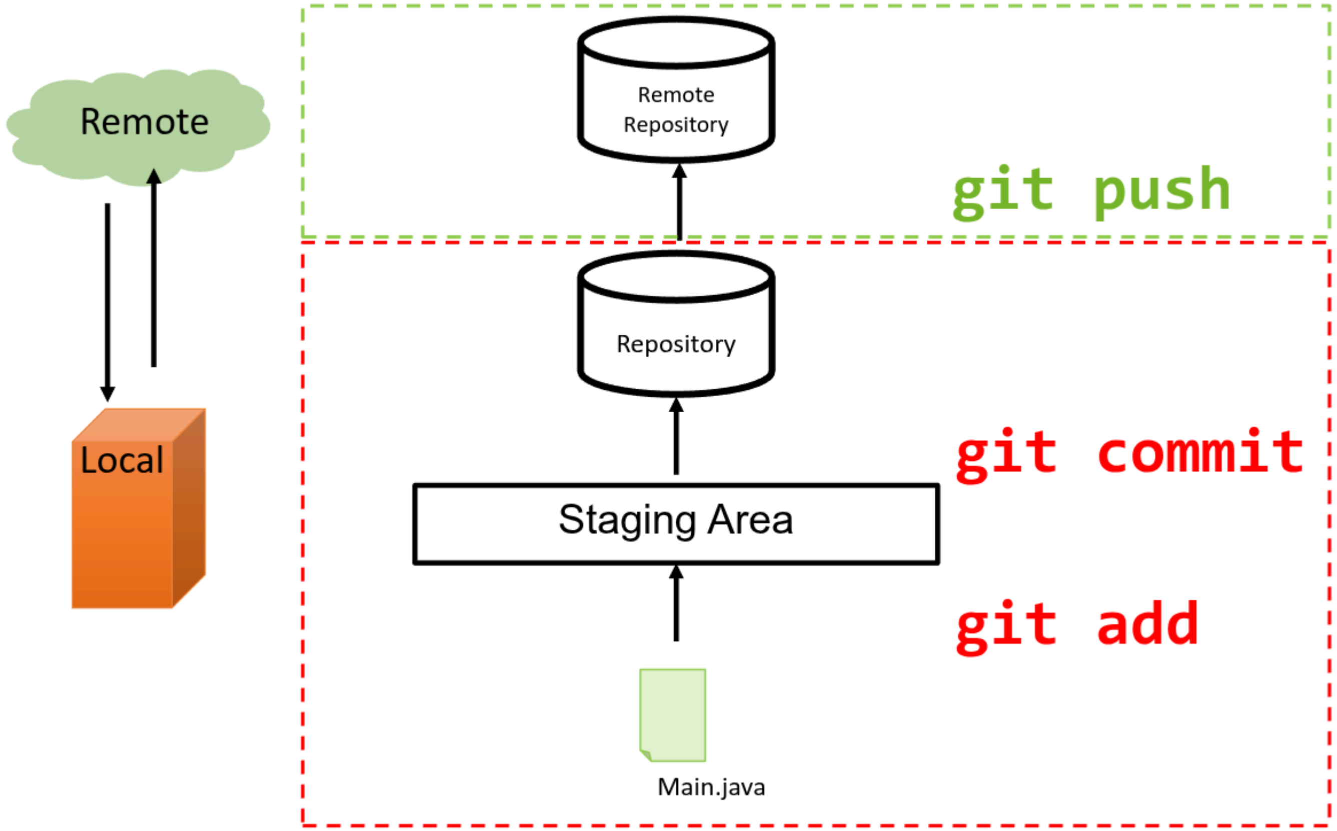
Remote Repository

- Server Computer
  - “**remote**” repository
  - e.g.: GitHub, GitLab, …

- Developers Computer A
  - “**local**” repository
  - Working directory

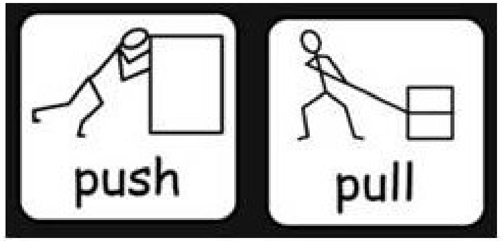


Working with remote repositories



Woring with other developers

- Version of local repository shared on the network (e.g. GitLab)
- Collaborate with other developers
  - The remote repository is necessary to simplify team collaboration.
  - A developer share latest commits / versions with **push** *(transfer data to server)*
  - To get changes (latest commits / versions) from other developers, a **pull** is done *(transfer data from server)*
- “It’s about pushing and pulling”



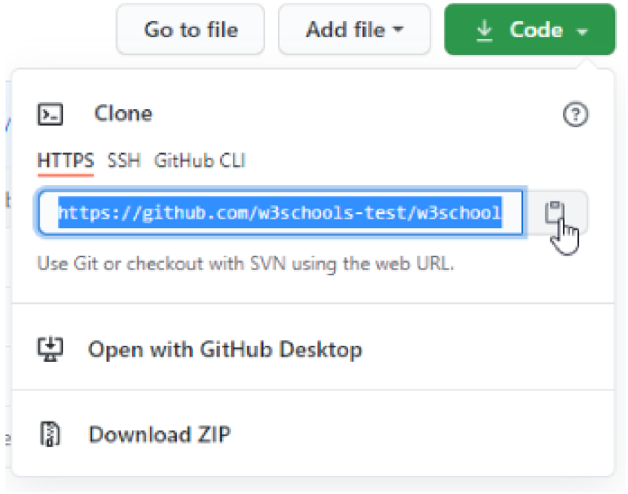
Source: vijaysangamworld.wordpress.com

Git “init” with remote repository

1. `git init`
2. `git add <filename>`
3. `git commit -m "initial commit"`
4. `git remote add origin <url>`
5. `git push --set-upstream origin main`

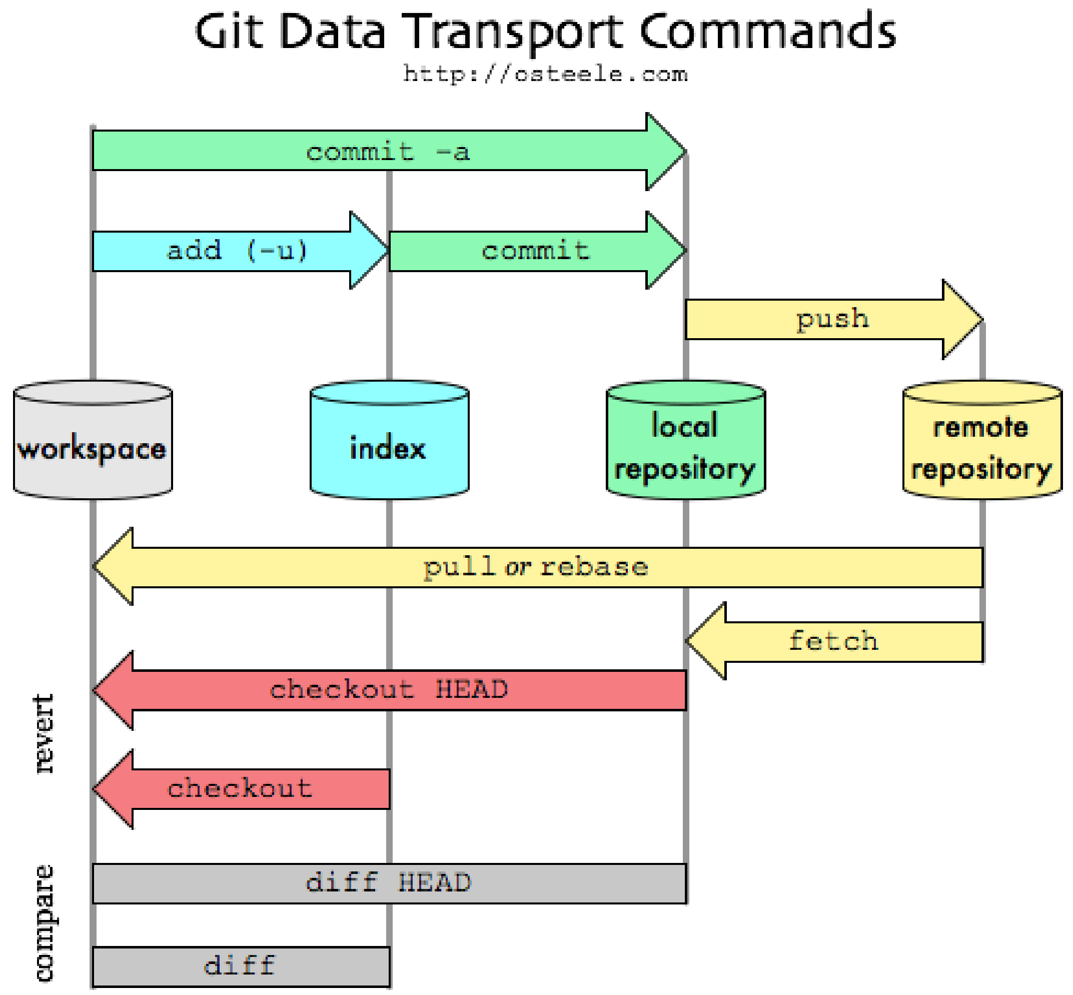
OR

1. `git clone <url>`



https://github.com/w3schools-test/w3school

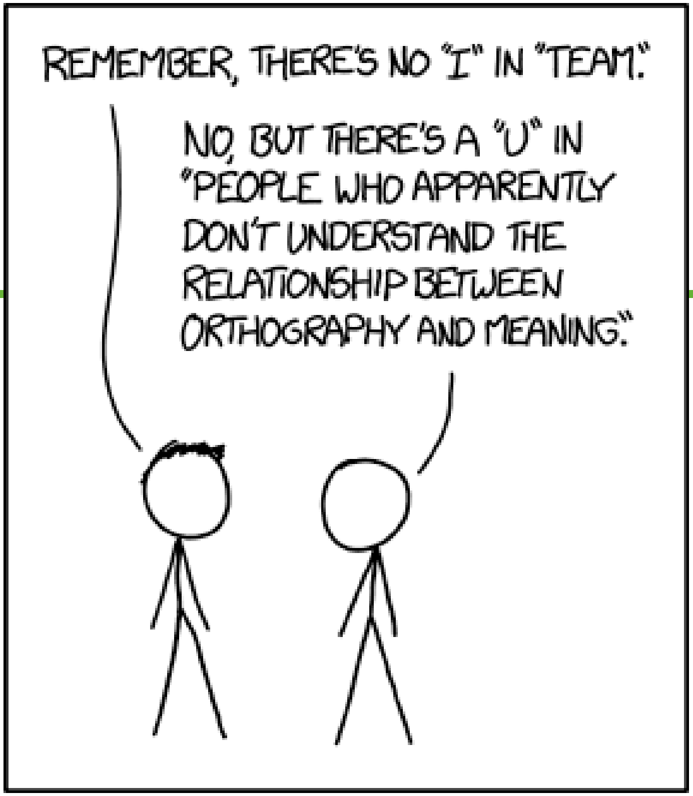
Use Git or checkout with SVN using the web URL.



- `git push`: pushes local changes to remote repository
- `git fetch`: fetches remote changes to local repository, but will **not** merge remote branches with local branches!
- `git pull`: fetch + merge in one command

- `git clone`: clones remote repository *once* to your local machine. Includes complete history

Team Work


REMEMBER, THERE'S NO "I" IN "TEAM."
NO, BUT THERE'S A "U" IN "PEOPLE WHO APPARENTLY DON'T UNDERSTAND THE RELATIONSHIP BETWEEN ORTHOGRAPHY AND MEANING."

Source: https://xkcd.com/1562/

First experience or problems

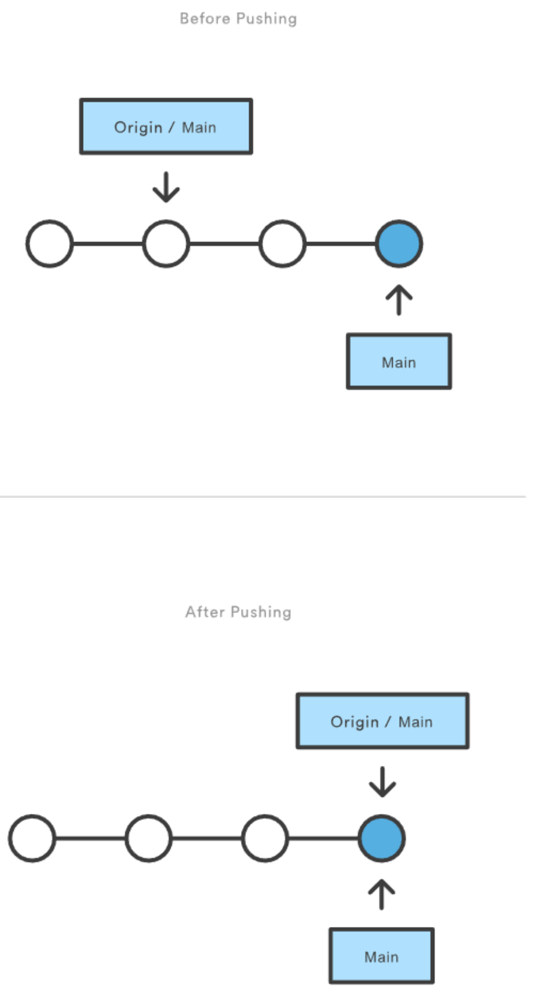
Source: https://devrant.com/rants/1211764/when-you-try-to-merge-branches-d

- Merge conflicts **constantly** occur when we work as a team.
- Today I was quicker with my push and it hit my colleague.
- Git is useless, there are constant conflicts.
- Who came up with that?

git push

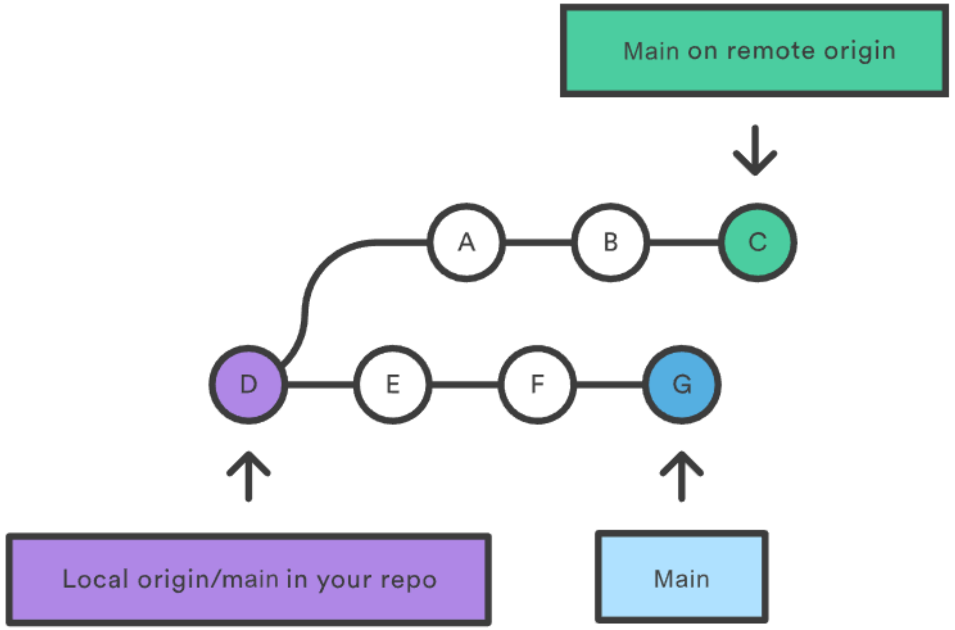
- After local changes, **push** is used to share modifications of current branch with team by uploading changes to remote repository

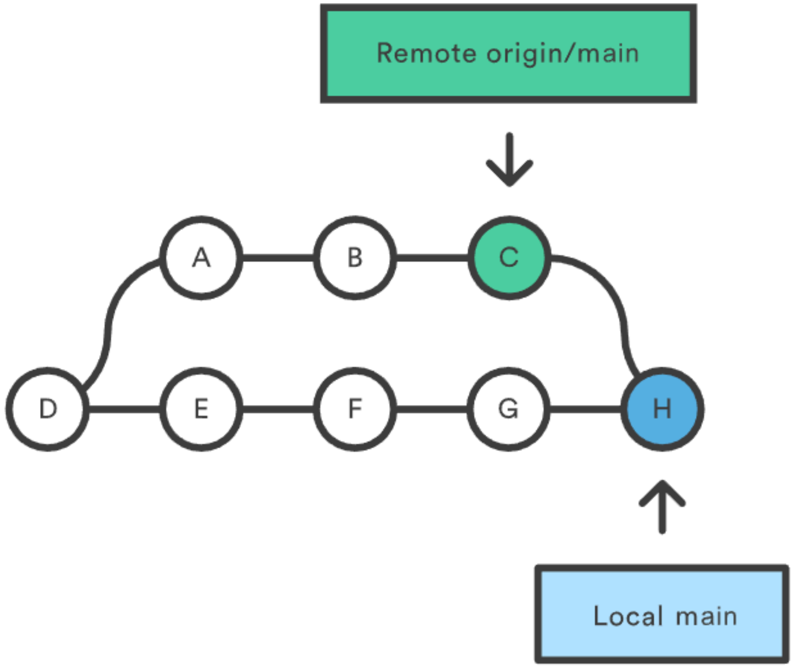
- `git push origin` *`branch_name`*
  - push local branch to remote repository
    - only once necessary



git pull

Fetch the remote version of the current branch and merge it into local one

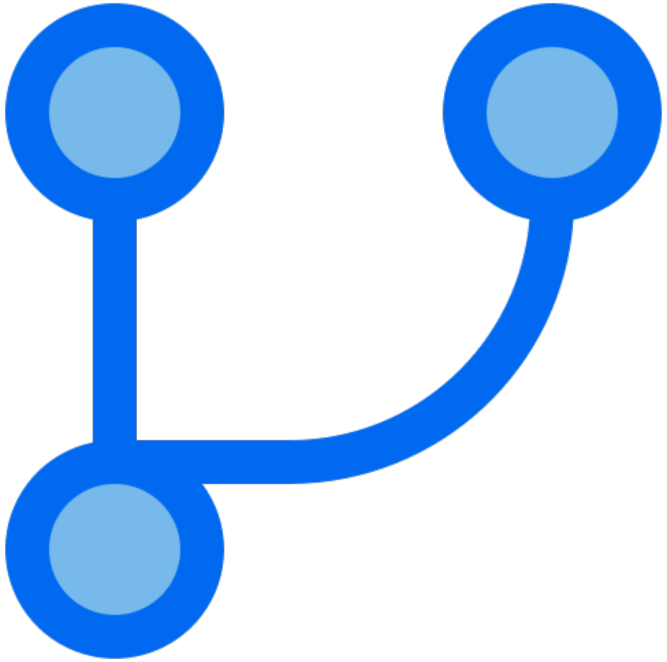




git branch

- When we never used a remote branch, we do not have a local version of it
  - Only after checkouts

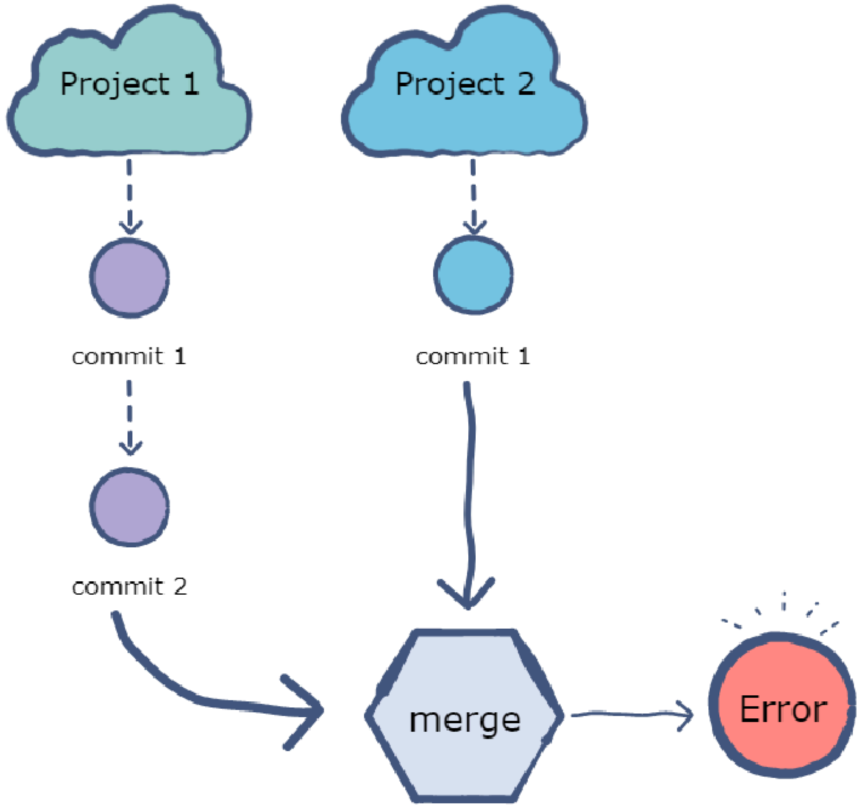
- `git branch -r`
  - shows remote branches
- `git branch -a`
  - shows local and remote branches



Problem: merge unrelated histories

- When multiple team member work on the same branch, this error will occur constantly because of different versions – various histories
  - “fatal: refusing to merge unrelated histories”

- possible fix
  - `git pull` + manual merge conflict every time



Better approach?

Use a **distributed workflow** to avoid many conflicts. Each Developer works on it’s own branch(es), merge is done centralized.

For example:

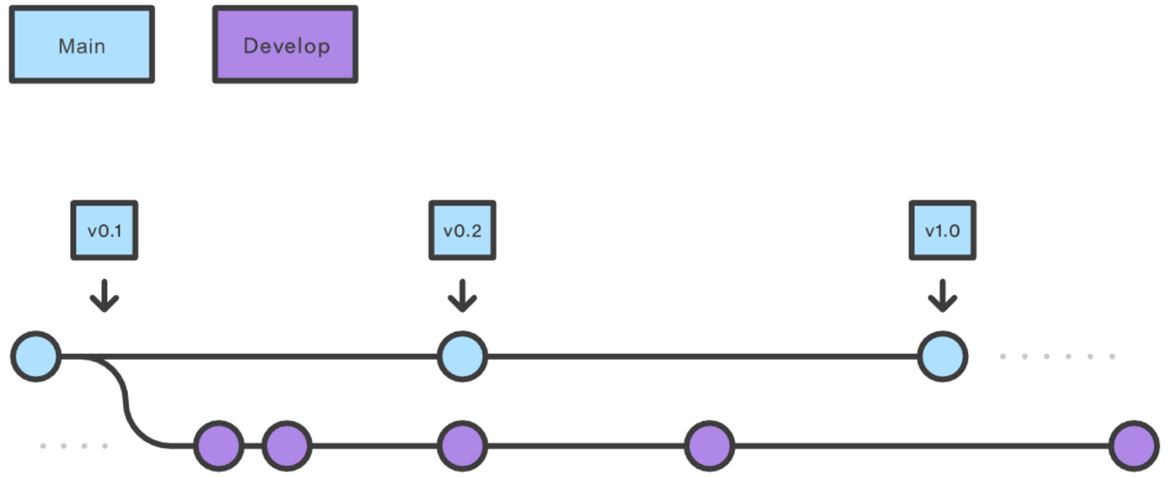
- Feature Branch Workflow
- Gitflow Workflow
- Forking Workflow
- …

Feature Branch Workflow

- All development should be realised in dedicated branch
  - *main* will never contain broken code
- Branch names should be descriptive
  - `issue-#1234` or `feat-menu`
- Feature branches can be pushed to central repository
- Each developer works on their dedicated branch(es)
- Changes will be discussed and merged via **pull request**

Gitflow Workflow

- Legacy flow with git branches
- Main branch stores official release history
- Develop branch is integration branch for features
- Merge is done by centralized member(s) or through *pull request*



Forking Workflow

- Every developer has own server-side repository
  - Not only one central repo
  - Often used in open-source projects
- Each contributor has
  - private local repo
  - public server-side one
- Developer push to their own server-side repo
  - Open **pull request** to “official” repository of maintainer
- Project maintainer can accept contributions without giving write access to project

References

- *Günther Popp*
  Konfigurationsmanagement
  dpunkt.verlag, 2008

- *Scot Chacon, Ben Straub*
  Pro Git
  Apress, 2nd Edition, 2014

- *Jon Loelinger, Matthew McCullough*
  Version Control with Git
  O'REILLY, 2012

- *Bernd Öggl, Michael Kofler*
  Git – Projktverwaltung für Entwickler und DevOps-Teams
  Rheinwerk Verlag, 2025

Links

- *Peter Cottle*
  LearnGitBranching
  https://learngitbranching.js.org/

- *GitHub*
  https://github.com/

- *GitLab*
  https://gitlab.com/

- *BitBucket*
  https://bitbucket.com/

- *Git*
  Git Source Code Management
  https://git-scm.com/

  git Book
  https://git-scm.com/book/en/v2/

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

All images used without sources are from https://www.flaticon.com/