Introduction to Git

Git: A distributed version control system

Advantages:

- Every repository has a full version history
- Most operations run locally
- Reliable data handling, ensuring integrity and availability
- Efficient data management for versions and branches
- Scalable collaboration mechanisms for large teams and complex projects

Caveats:

- Need to learn and understand the underlying model
- Not built for binary files or large media files

Learning objectives

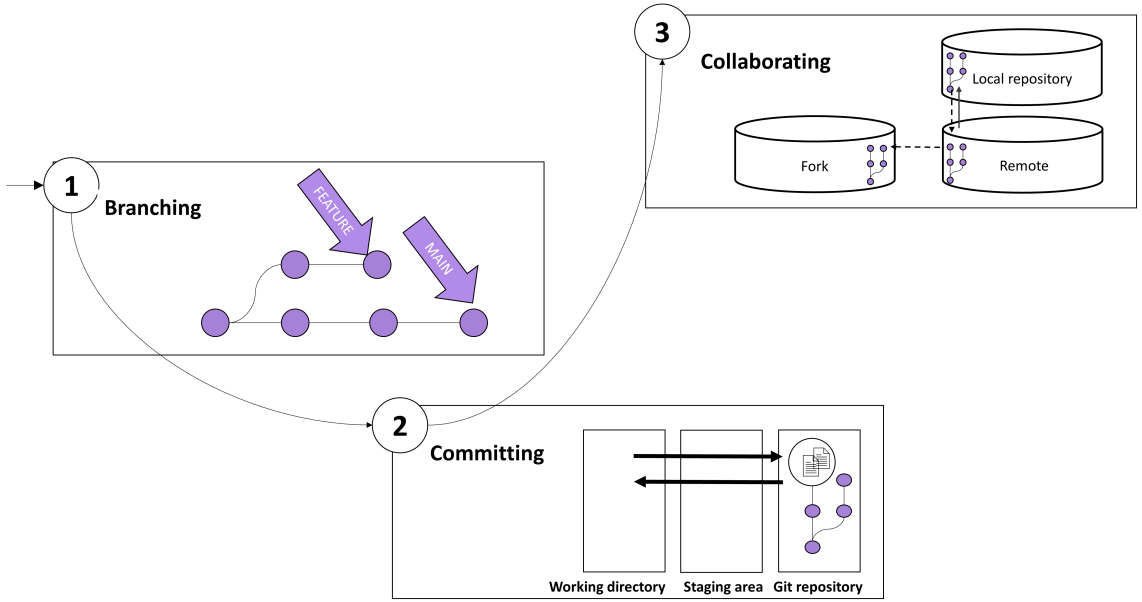
Understand and use git to develop software in teams.

Part 1: Branching
Part 2: Committing
Part 3: Collaborating

Each part starts with the **concepts** before the **practice** session.

In the practice sessions:

- Form groups of two to three students
- Work through the exercises
- Create a **cheat sheet** summarizing the key commands

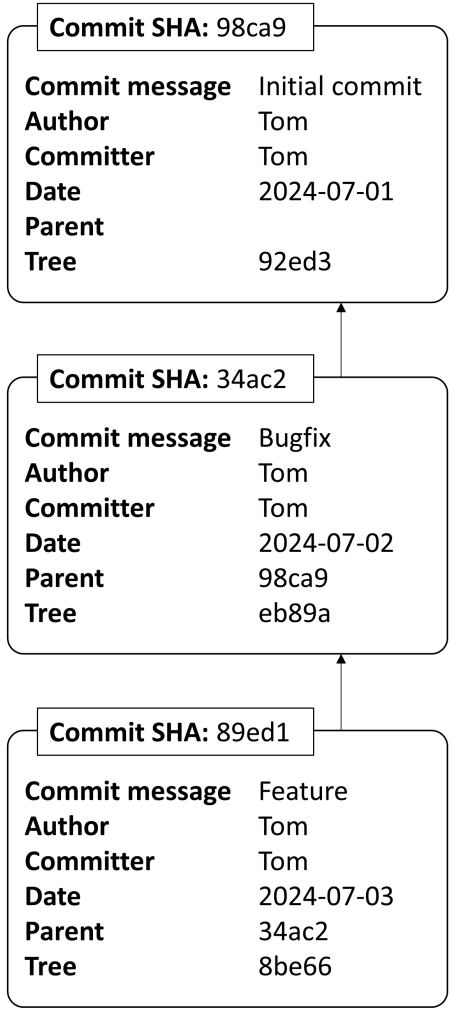


> * Note: This session is based on our unique and peer-reviewed approach.

Part 1: Branching

Commits

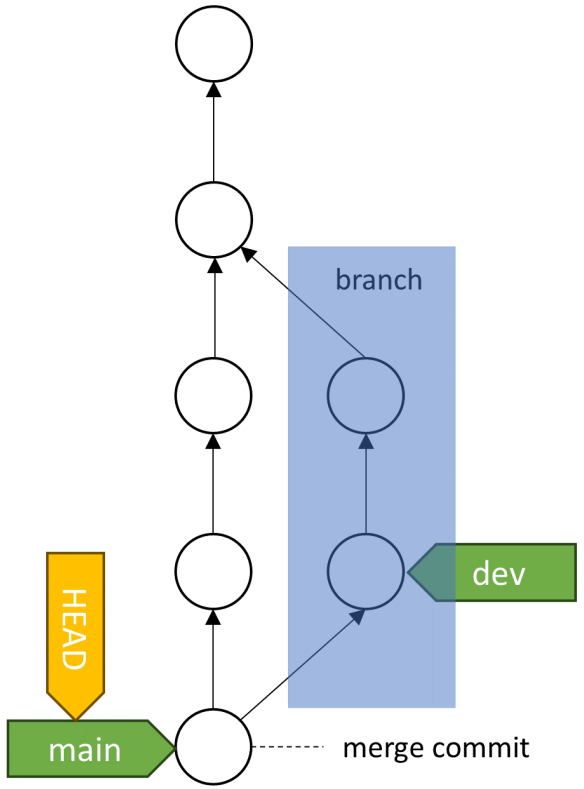
- A **commit** refers to a snapshot (version) of the whole project directory, including the meta data and files
- Commits are identified by the **SHA** fingerprint of their meta data and content*, e.g., `98ca9`
- Commits are created in a sequence, with every commit pointing to its **parent** commit(s)
- The **tree** object contains all files (and non-empty directories); it is identified by a SHA hash
- Commits are created by the **git commit** command



> * If any of the meta data or content changes, the SHA will be completely different.

The DAG, branches, and HEAD

- Commits form a **directed acyclic Graph (DAG)**, i.e., all commits can have one or more children, and one or more parents (except for the first commit, which has no parent). Closed directed cycles are not allowed.
- With the **git branch <branch-name>** command, a separate line of commits can be started, i.e., one where different lines of commits are developed from the same parent. The branch pointer typically points at the latest commit in the line.
- With the **git switch <branch-name>** command, we can select the branch on which we want to work. Switch effectively moves the HEAD pointer, which points to a particular branch and indicates where new commits are be added.
- With the **git merge <other-branch>** command, separate lines of commits can be brought together, i.e., creating a commit with two parents. The *merge commit* integrates the contents from the *<other-branch>* into the branch that is currently selected. The *<other-branch>* is not changed.

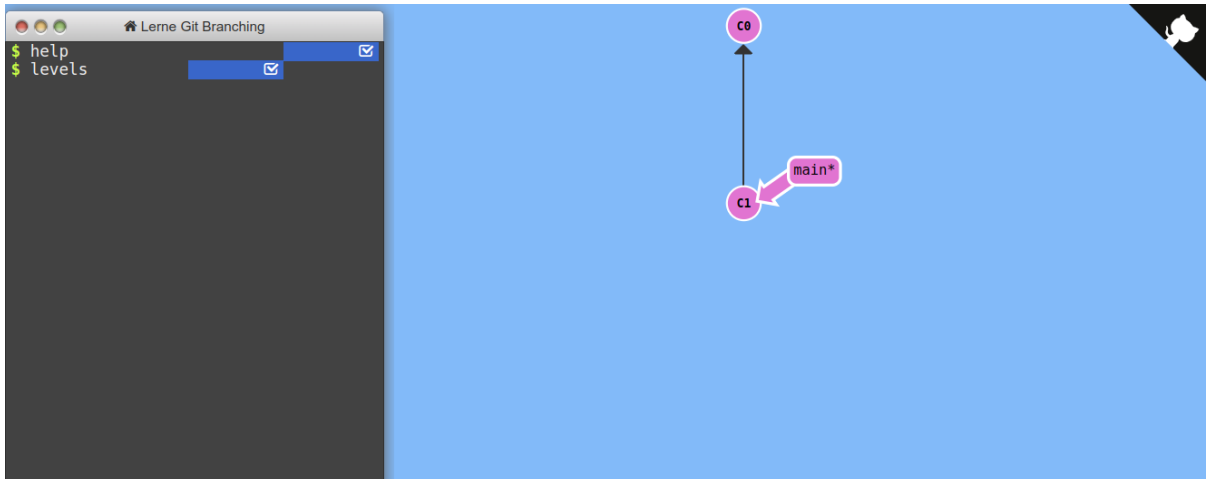


> Note: Arrows point from children to parent commits.

Practice: Branching

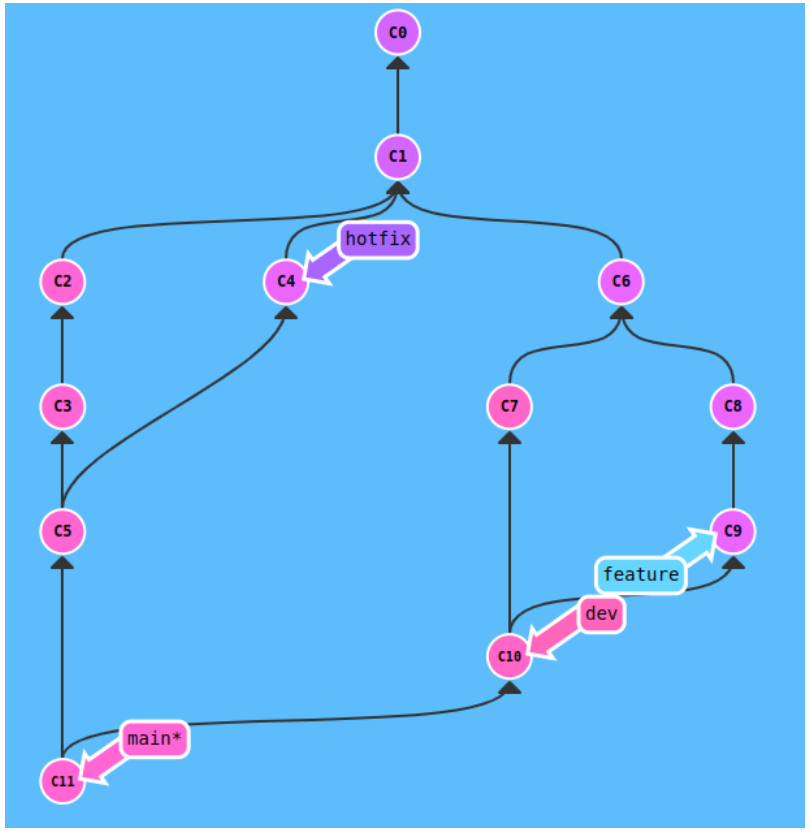
To practice git branching, we use the learn-git-branching tutorial.

Complete the first two levels on branching, merging, and navigating in the git tree.



NOTE: You can type "undo" when you made a mistake.

To continue practicing, create the following tree, which resembles a typical setup of git branches.



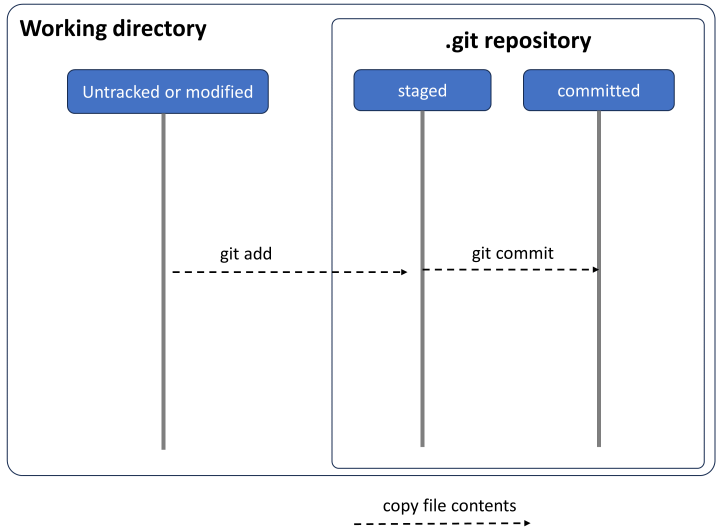
Part 2: Committing

The working directory and .git repository

All working file contents reside in the working directory; staged and committed file contents are stored in the `.git` directory (a subfolder of the working directory).

Git allows us to stage (select) specific file contents for the next commit.

- With **git add <file-name>**, contents of an *untracked or modified* file are copied to the `.git` repository and added to the staging area, i.e., explicitly marked for inclusion in the next commit.
- With **git commit**, *staged* files contents are included in a *commit*.

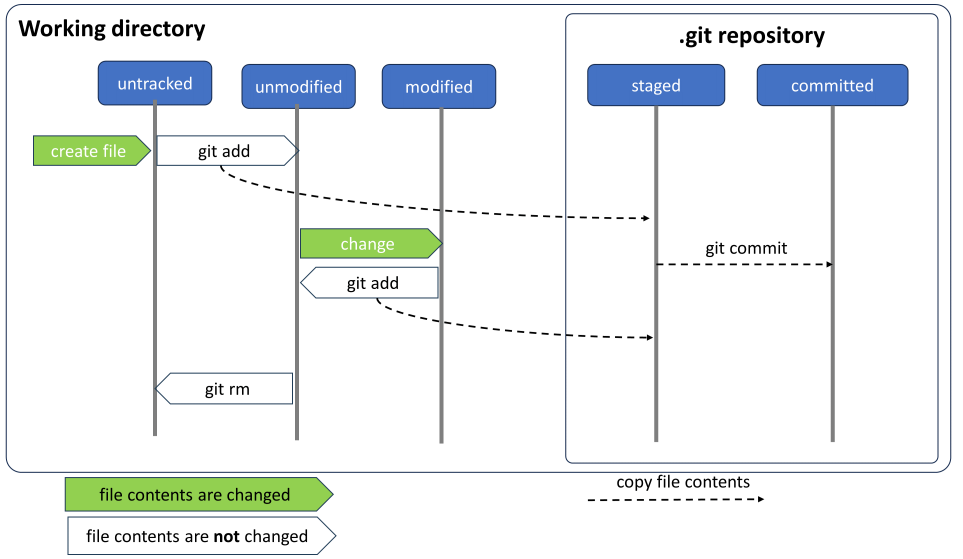


The three states of a file

Files in the working directory can reside in three states:

- New files are initially **untracked**, i.e., Git does not include new files in commits without explicit instruction.
- With *git add*, file contents are **staged** and the file is **tracked**. Given that the file in the working directory is identical with the staged file contents, the file is **unmodified**.
- When users change a file, it becomes **modified**, i.e., the file in the working directory differs from the file contents in the staging area.
- With *git add*, the file contents are staged again, and the file becomes **unmodified**.
- With *git rm*, files are no longer tracked.

Note: *git add* and *git rm* do not change the contents of the file in the working directory.



Undoing changes

Git status

add screenshot?

TODO : diff and contents

- For convenience, Git displays the diff of file contents.

When writing code, we can make mistakes, or we may need to undo or modify previous changes. To undo changes, it is important to understand whether they are unstaged, staged, or committed.

To **undo unstaged or staged changes**, `git status` suggests the corresponding operations (`git restore <file>` and `git restore --staged <file>`). To see how `git restore` works:

- Modify the `README.md` file and add the changes to the staging area
- Undo the staged changes
- Undo the unstaged changes

Check: The working directory should be clean again.

To **undo committed changes**, there several options (some are available in gitk):

- Revert the commit, i.e., create a new commit to undo changes: `git revert COMMIT_SHA --no-edit`
- Undo the commit and leave the changes in the staging area: `git reset --soft COMMIT_SHA` (*)
- Stage changes, and run `git commit --amend` to modify the last commit (*)

If you have the time, try the different undo operations in the session.

(*) Important: only amend commits that are not yet shared with the team. Otherwise, revert is preferred.

Transfer challenges I

1. Consider how the **git switch** (or the revert/pull/checkout) command affects the git areas. How does it affect the working directory?
2. Git provides the option to edit prior commits using an interactive rebase, such as the **git rebase -i**. How would that affect the following commits?

Illustration: Git Merge

Setting: Two authors working on the same document (paper.md).

1. Setup the code skeleton
2. Write different parts of the same document

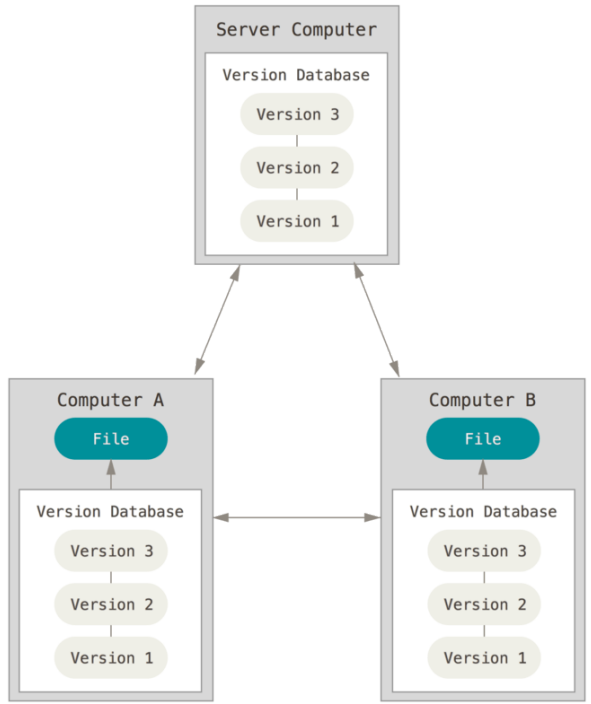
- git checkout -b author_1 (add introduction)
- git switch main & git checkout -b author_2 (add background)
- git switch main & git merge author_1 (fast forward)
- git merge author_2 (merge commit, no conflict)

3. Edit the same part (remember to merge both branches with main)

- case 1: conflicting contents that contradict each other
- case 2: conflicting contents that need to be resolved

Part 3: Collaborating

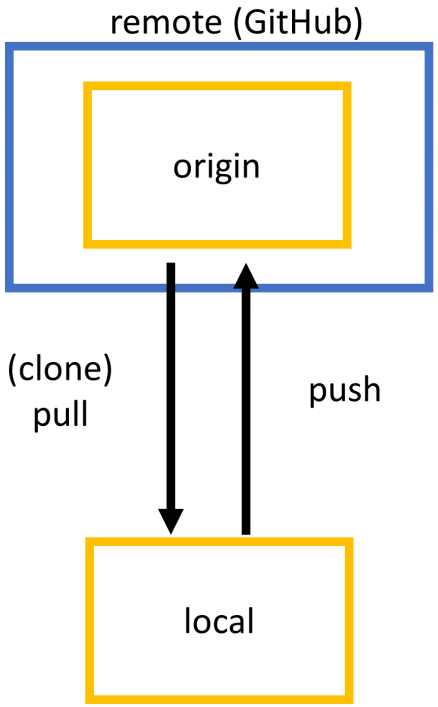
TODO:


Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Version 3
Version 2
Version 1
Computer B
File
Version Database
Version 3
Version 2
Version 1

Remote collaboration

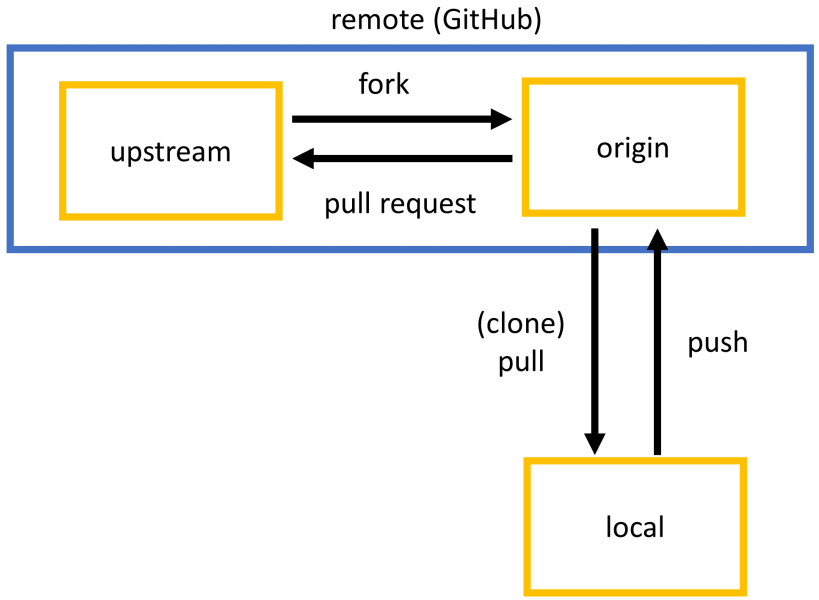
- To collaborate, a remote repository is needed (named origin)
- If the remote repository exists, the **git clone** command retrieves a local copy
- If the remote repository does not exist, you have to add the remote origin and push the repository
- To retrieve changes, use the **git pull** command
- To share changes, use the **git push** command

This model works if you are a maintainer of the remote/origin, i.e., if you have write access.



Forks

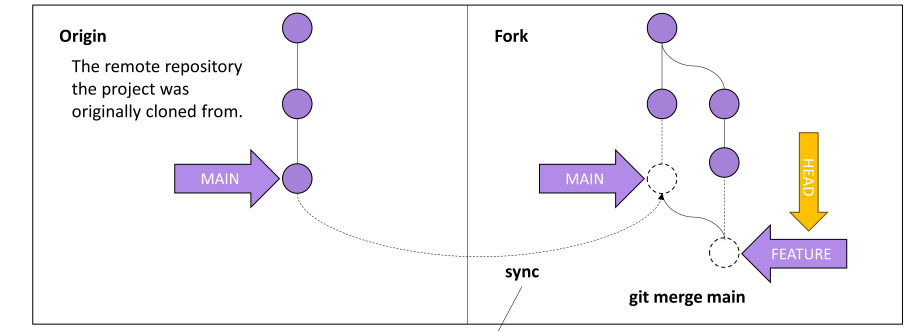
- In Open-Source projects, write-access is restricted to a few maintainers
- At the same time, it should be possible to integrate contributions from the community
- **Forks** are remote copies of the upstream repository
- Contributors can create forks at any time and push changes
- Contributors can open a **pull request** to signal to maintainers that code from the fork can be merged
- Pull requests are used for code review, and improvements before code is accepted or rejected

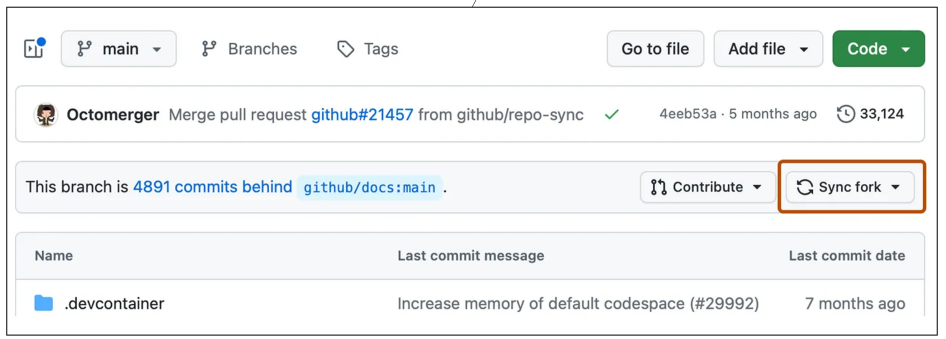


Fork, invite, clone, and pull-request on GitHub

Work in a forked repository

- In the fork, it is recommended to create working branches instead of committing to the `main` branch.
- It is good practice to regularly **sync** the `main` branches (on GitHub), and merge the changes into your working branches (locally or on GitHub).
- Syncing changes may be necessary to get bugfixes from the original repository, and to prevent diverging histories (potential merge conflicts in the pull request).





Remotes and branches

- Most remote operations, including pull, push, pull requests refer to branches
- In some cases, **branches must be selected explicitly**: pull requests, or pulling new branches
- In other cases, git automatically selects branches, i.e., it remembers the typical branch to pull or push