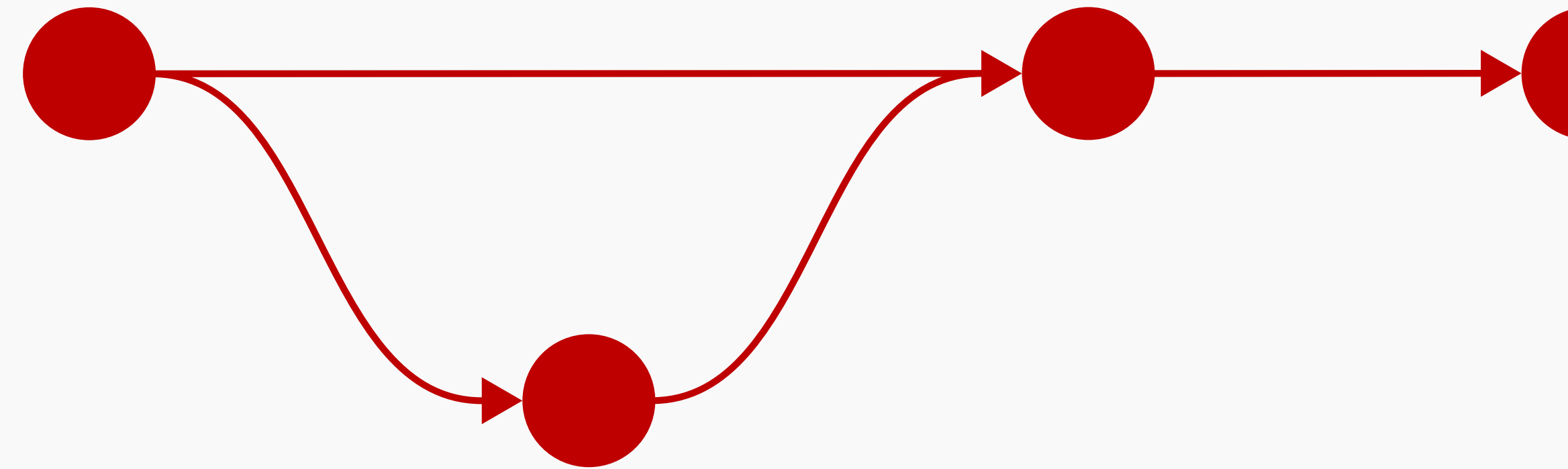


# Introduction to Git

Presented by

Robben Migacz

Scientific Consultant



## Please note

We will be using Git from the command line. If you are not familiar with the **basics** of navigating the Linux command line (e.g., `cd`, `ls`, a text editor), we recommend going through our introductory Linux training sessions first.



*We won't be doing anything too complicated,  
so you can probably follow along even if you  
don't have experience using the command line*



## **Please note**

We will be using Git from the command line. If you are not familiar with the **basics** of navigating the Linux command line (e.g., **cd**, **ls**, a text editor), we recommend going through our introductory Linux training sessions first.

## Please note

You can use Git from the command line on the Center for High Performance Computing's Linux resources (the clusters). **If you do not have a CHPC account, however, please download Git so you can follow along.**





# Installing Git

- **Windows:** Download *Git for Windows* from your browser
- **macOS:**
  - Run `git` from the Terminal application, which will prompt you to install it
  - Alternatively, Git is also available through Homebrew or MacPorts
- **Linux:**
  - Use Git on CHPC resources with `module load git`
  - `sudo apt install git` (Debian, Ubuntu, Linux Mint, etc.) or `sudo dnf install git` (Red Hat Enterprise Linux, Fedora Linux, Rocky Linux, etc.)

... or get Git *via* Git: `git clone https://github.com/git/git` 🐔? 🥚?

# What is Git?

Git is **version control software**. It helps you keep track of different versions of your files.



# What is Git?

Git is **version control software**. It helps you keep track of different versions of your files.



my\_file



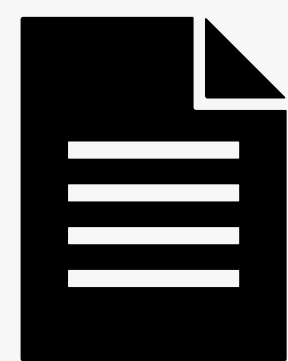
my\_file\_2



my\_file\_3



my\_file\_fixed



my\_file\_final



my\_file\_final\_  
final



my\_file\_FINAL

***Does this look familiar?***

Consider using Git to manage versions!

# What is Git?

Git is **version control software**. It helps you keep track of different versions of your files. This is particularly helpful when you are working on a project with other people, as it can help your team stay organized.

# What is Git?

**Git is not the same thing as GitHub.** The two are often conflated by new users. GitHub is a web platform that is used to store, share, and work on Git repositories. (Learning Git will help you understand GitHub, so don't close the window *just yet* if this isn't quite what you were expecting.)

Git is open-source software. It is not owned or maintained by GitHub. As of Fall 2024, its source code has more than 1,700 contributors.



# Who uses Git?

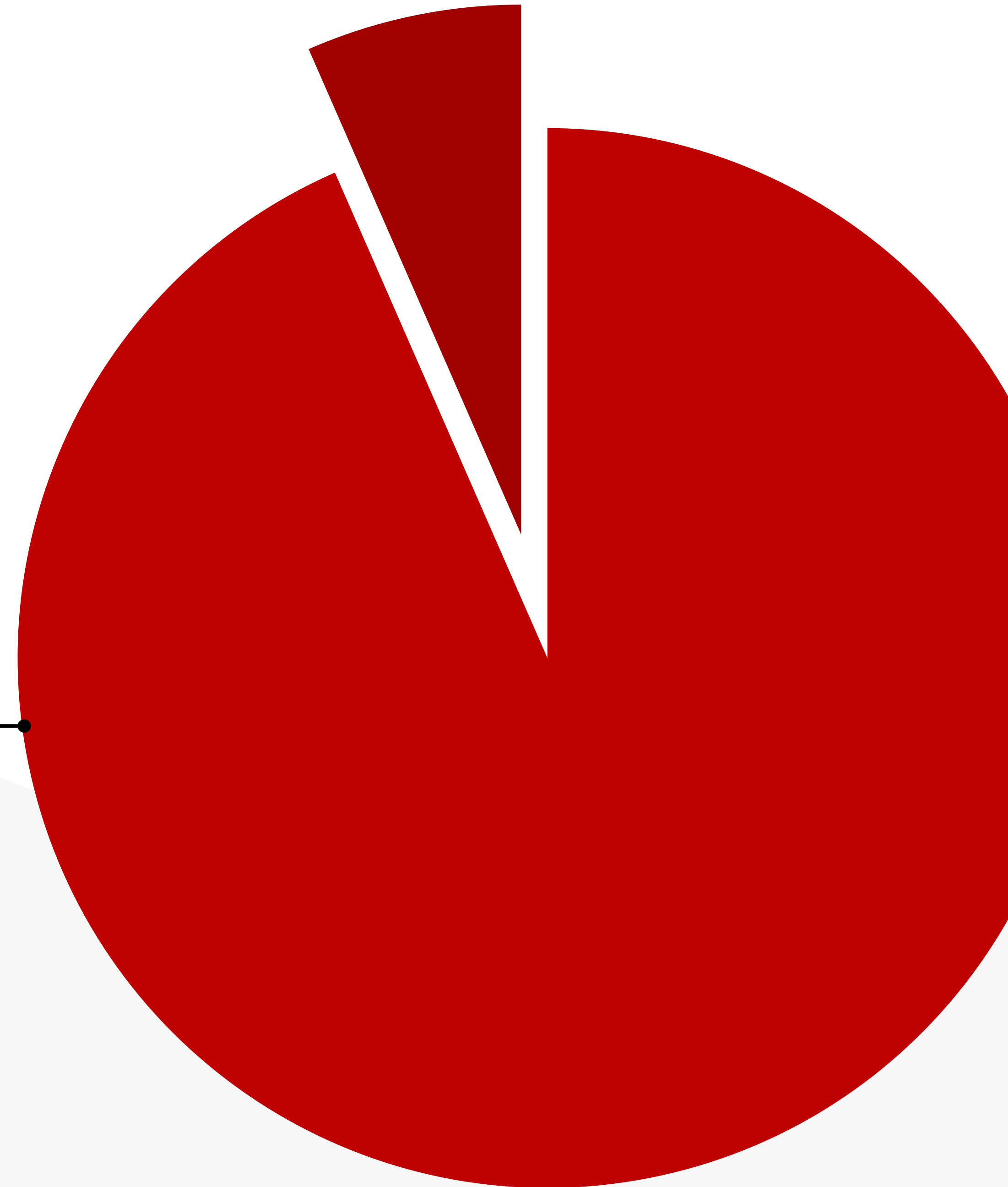
Git is used primarily by **software developers**.



# Who uses Git?

Git is used primarily by **software developers**.

In the Stack Overflow Developer Survey 2021, more than 93% of respondents said they use Git.

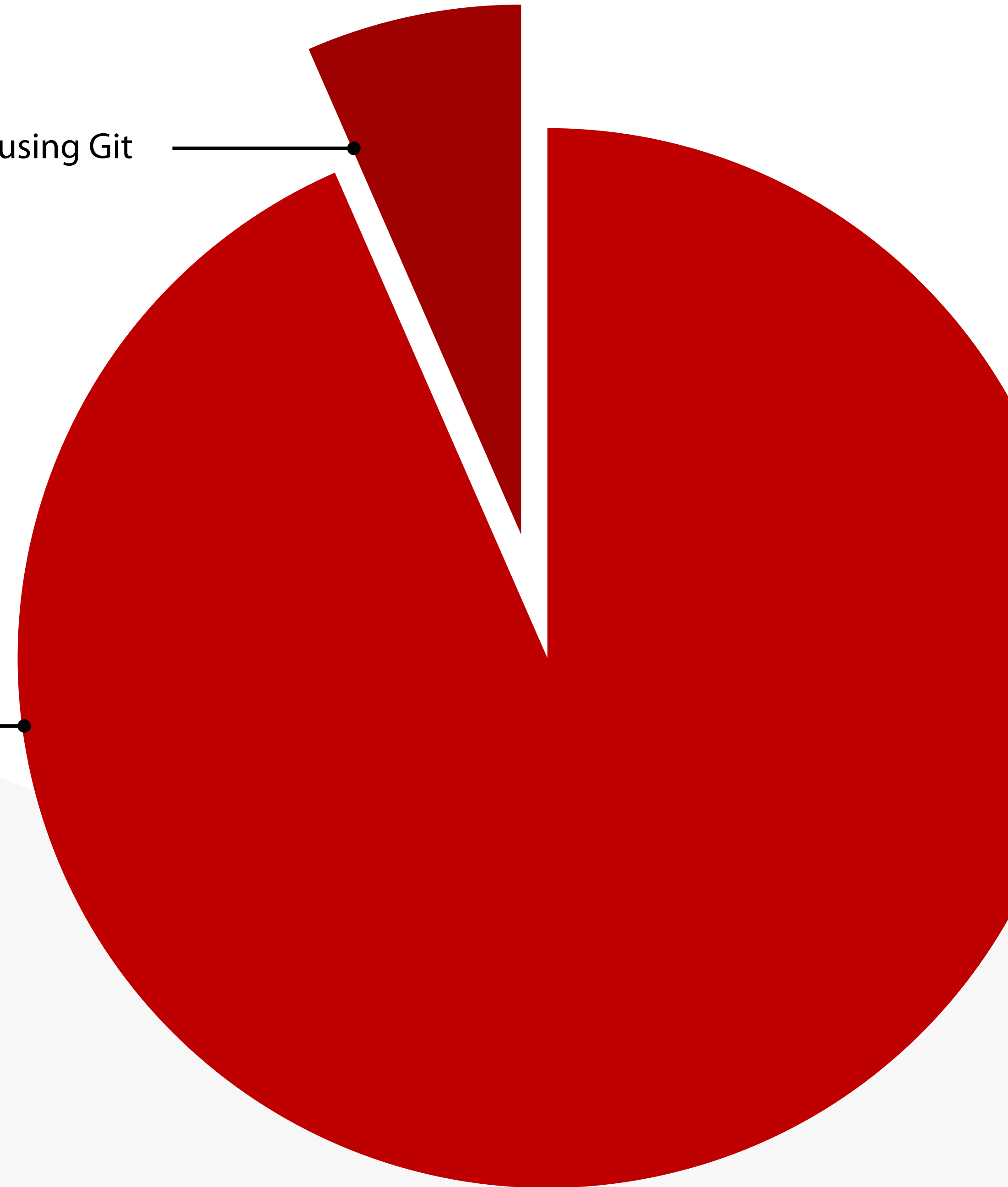


# Who uses Git?

Developers who should probably start using Git

Git is used primarily by **software developers**.

In the Stack Overflow Developer Survey 2021, more than 93% of respondents said they use Git.





# However, Git isn't *just* for software developers

Git will help you keep track of changes to *any* of your files. It works best with plain text (anything human-readable in a text editor) and relatively small files.

It works well with LaTeX documents, for example.



**Let's learn how to use Git!**





Git ... frustrates me because, while it is an excellent productivity tool that lets scores of developers collaborate on a single code base without clobbering each other, it asks its users to understand how it works at its deepest levels. It's like if you were going to write a letter and first had to read a treatise on UTF-8 encoding.

— Sharon Cichelli, “Git is a Directed Acyclic Graph and What the Heck Does That Mean?”, 2017

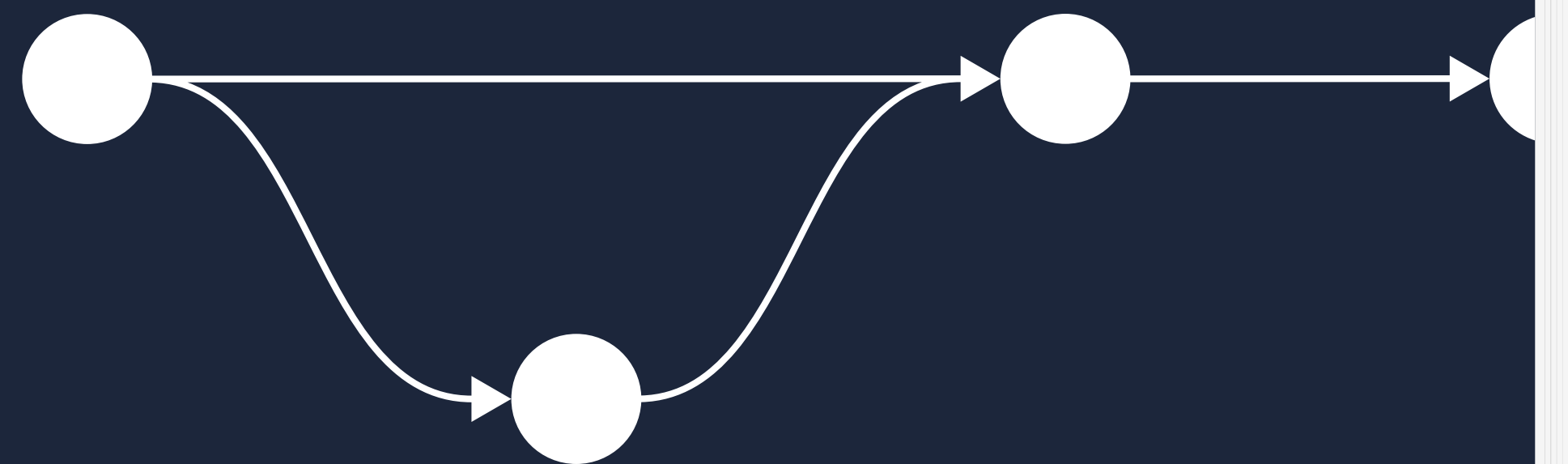


We'll need to cover some terminology before we can move into the hands-on material.

Git can be difficult to understand at first; taking a step back to see the big picture will help when we start looking at commands.

## Huge Glossary of Git Vocabulary

YES, YOU NEED TO KNOW THIS



## Terminology: Repository

A **repository** is where the revision history of a project is stored. It's a hidden directory, `.git`, within the directory your project resides in.

The `.git` directory's name starts with a period, which means it's "hidden." Depending on your file browser's settings, you may not be able to see it. You probably won't need to; we'll use commands to interact with the repository.

Just know that your project history doesn't live in a mysterious database or "in the cloud." It's just a collection of files in the same directory as your project.



## Terminology: Repository

There are a few important things to note about repositories:

## Terminology: Repository

There are a few important things to note about repositories:

- If you delete the .git directory, you lose your project history (and you are stuck with the current state of your files)



## Terminology: Repository

There are a few important things to note about repositories:

- If you delete the `.git` directory, you lose your project history (and you are stuck with the current state of your files)
- If you make a backup copy of your project directory, you also make a backup copy of your Git repository (contained within the project directory)
  - We always recommend backing up your important files!



## Terminology: Repository

There are a few important things to note about repositories:

- If you delete the `.git` directory, you lose your project history (and you are stuck with the current state of your files)
- If you make a backup copy of your project directory, you also make a backup copy of your Git repository (contained within the project directory)
  - We always recommend backing up your important files!
- If you send your project directory, including `.git`, to another person, the other person can see your project history

## Terminology: Repository

There are a few important things to note about repositories:

- If you delete the `.git` directory, you lose your project history (and you are stuck with the current state of your files)
- If you make a backup copy of your project directory, you also make a backup copy of your Git repository (contained within the project directory)
  - We always recommend backing up your important files!
- If you send your project directory, including `.git`, to another person, the other person can see your project history
- You should avoid storing sensitive information or passwords in a Git repository, especially if there is a possibility that you might share your work with others or use a host like GitHub



## Terminology: Commits

Git *does not* keep track of every change you make; it is *not* the same as having an undo button or a file history. Instead, you **commit** changes as you see fit.



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.

— *Pro Git* (2<sup>nd</sup> Edition), Scott Chacon and Ben Straub, 2014

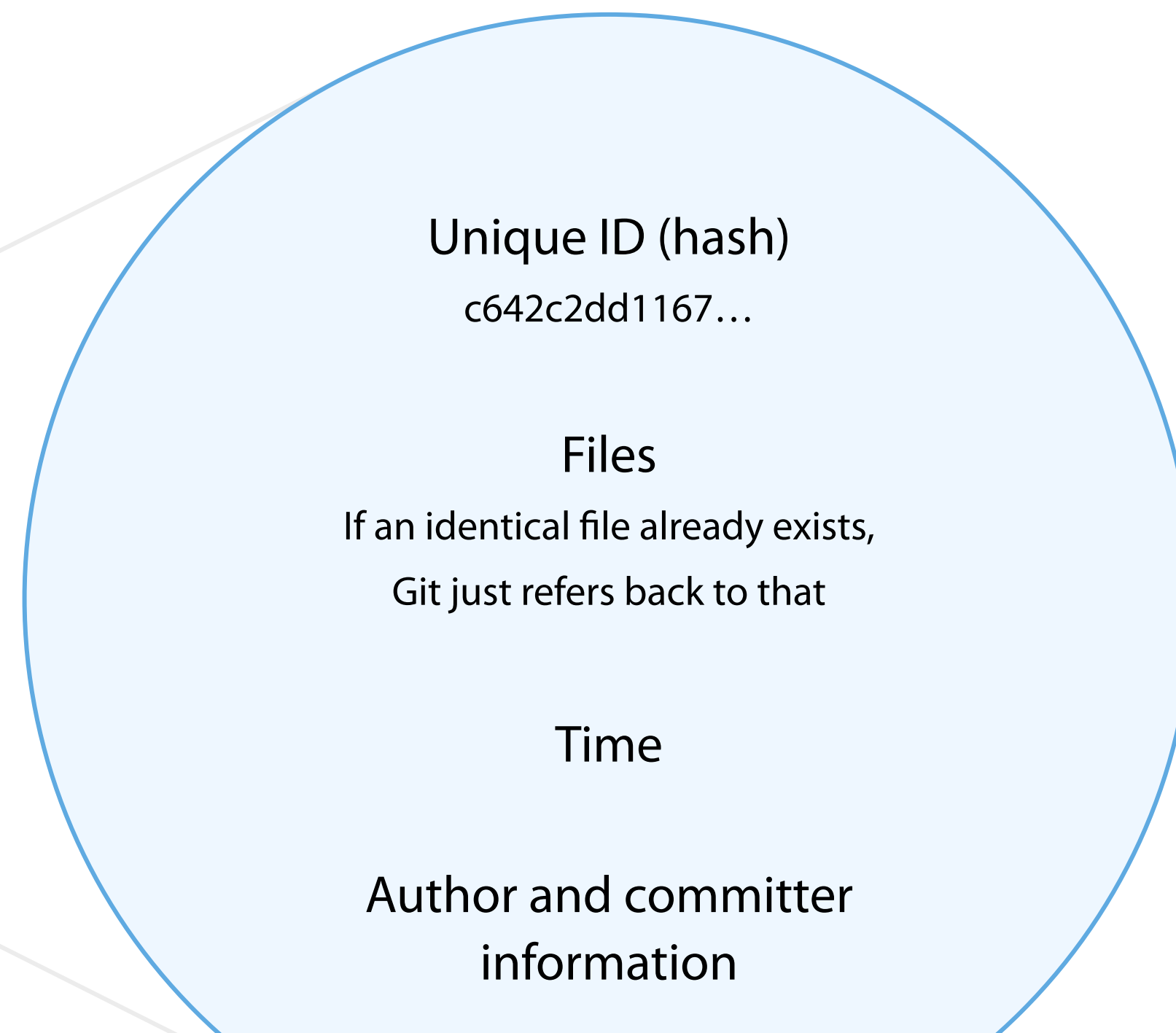
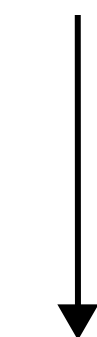


## Terminology: Commits

Commits have metadata such as the committer, the author (usually the same as the committer, but not always), the time, and a message describing the changes.

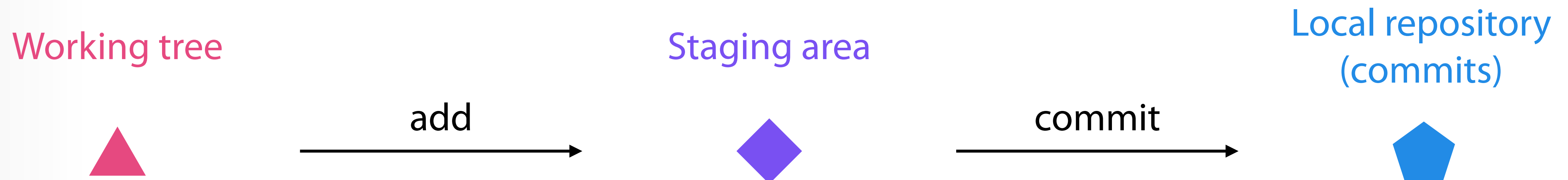
Every commit has an associated hash, which is the name of the commit according to Git. This is how you reference a specific commit. You can truncate (shorten) the hash in commands.

Commits are often drawn as a point or circle



## 📖 Terminology: Working tree and staging area (index)

Before you can make a commit, you need to tell Git *which* changes you want it to keep track of. You first work on files in the **working tree**. You then move files with changes you want to commit to the **staging area** or **index**. Only changes in the staging area are included in commits.



## 📖 Terminology: Graph

Together, commits form a (directed acyclic) **graph**.

- **Graph:** Commits (vertices) are related (connected by edges)
- **Directed:** Commits refer back to other commits (earlier versions); there is a direction inherent to relationships between commits
- **Acyclic:** If you follow the graph from a commit, that commit will not appear again; it cannot be its own “parent” (no matter how many generations removed)
  - In other words, a version of your project cannot be based on itself
  - There are no loops in the graph



**Let's apply these concepts and start using Git!**





# In practice: Getting help

Git comes with documentation that can help you learn more about each command. To get help with `git commit`, for example, run

```
git help commit
```

Additionally, there are a few built-in tutorials and a glossary:

```
git help tutorial
```

```
git help everyday
```

```
git help workflows
```

```
git help glossary
```

# In practice: Repositories and configuration

You can initialize a Git repository in any directory by running `git init`

You can configure your username, email address, and preferred text editor with `git config`.

```
git config --global user.name "Your Name"
```

```
git config --global user.email "your.name@your.domain"
```

```
git config --global core.editor "nano"
```

# In practice: Repositories and configuration

You can initialize a Git repository in any directory by running `git init`

You can configure your username, email address, and preferred text editor with `git config`.

```
git config --global user.name "Your Name"
```

```
git config --global user.email "your.name@your.domain"
```

```
git config --global core.editor "nano"
```

*--global sets a default value; you can omit it to apply a configuration to the current repository (but not other repositories)*



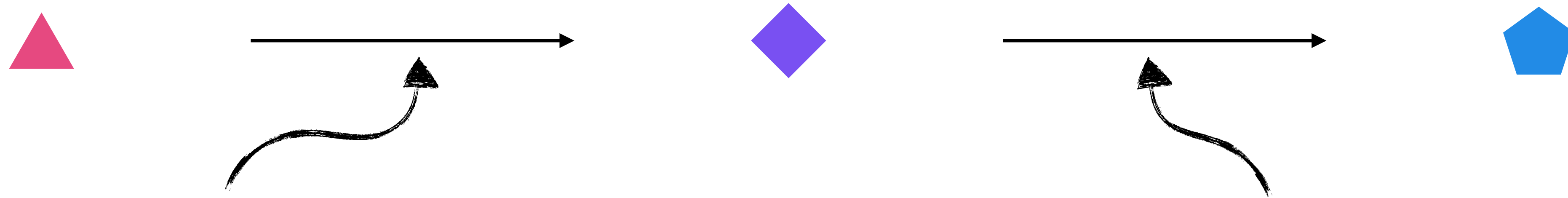


# Hands-on: Repositories and configuration

1. Create a new directory and initialize a Git repository in it
2. Configure your name, email, and text editor through Git
3. Run **git status** to check that the repository exists

*This command will show you which files are in the staging area and which have been modified in the working tree.*

# In practice: Commits



You need to *add* files to the *staging area* before you *commit* changes.

```
git add filename
```

```
git add *.py
```

```
git add .
```

```
git add --all
```

You can remove files from the staging area with

```
git rm --cached filename
```

# In practice: Commits

Once you have added your changes to the staging area, you can create a new commit with

```
git commit
```

This will open a text editor for a commit message. Alternatively, use

```
git commit -m "Your message here"
```

to specify a message without opening a text editor.



# In practice: Logs and differences

Once you've committed your changes, you can see them in the log:

```
git log --graph --all
```

Additionally, you can compare two different project states:

```
git diff earlier-commit later-commit
```

```
git diff --word-diff=color earlier-commit later-commit
```

Without *earlier-commit* and *later-commit*, this will compare the current state to the previous commit.

# Hands-on: Commits

1. Create a file in your project directory
2. Add your file to the staging area with `git add`
3. View the repository status with `git status`
4. Create a commit with `git commit`



# Hands-on: Commits

1. Create a file in your project directory
2. Add your file to the staging area with `git add`
3. View the repository status with `git status`
4. Create a commit with `git commit`
- ★ 5. Modify your file
6. Repeat (2), (3), and (4)
7. Use `git log` to see what your project history looks like; try a `git diff` between commits

## Terminology: Distributed version control

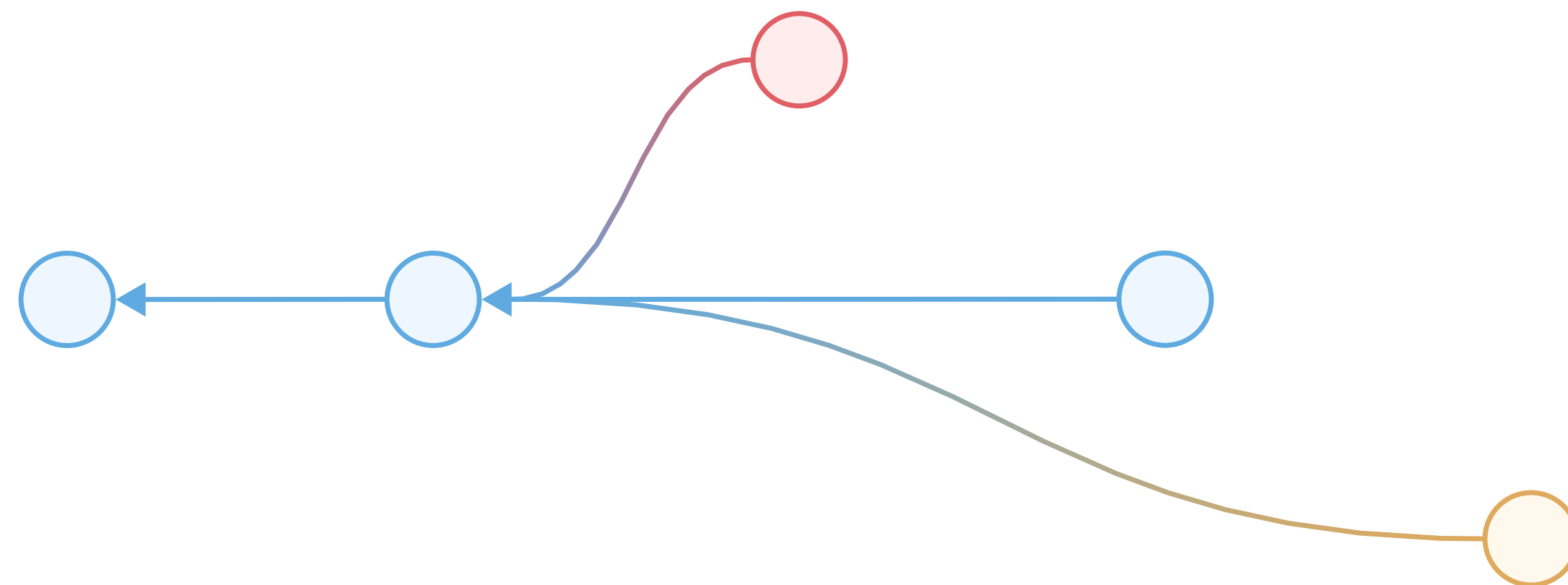
Git is a **distributed** version control system. Everyone working on the project has a copy of the repository. There is no (required) single source of truth or central server.

*Everyone can always edit every file.* Git makes no effort to prevent users from modifying the same parts of the project at the same time (often limited by needing to “check out” and “check in” files with a central server in other version control systems). Instead, users may need to manage conflicts. This isn’t as chaotic or scary as it sounds, though!

## 📖 Terminology: Branches

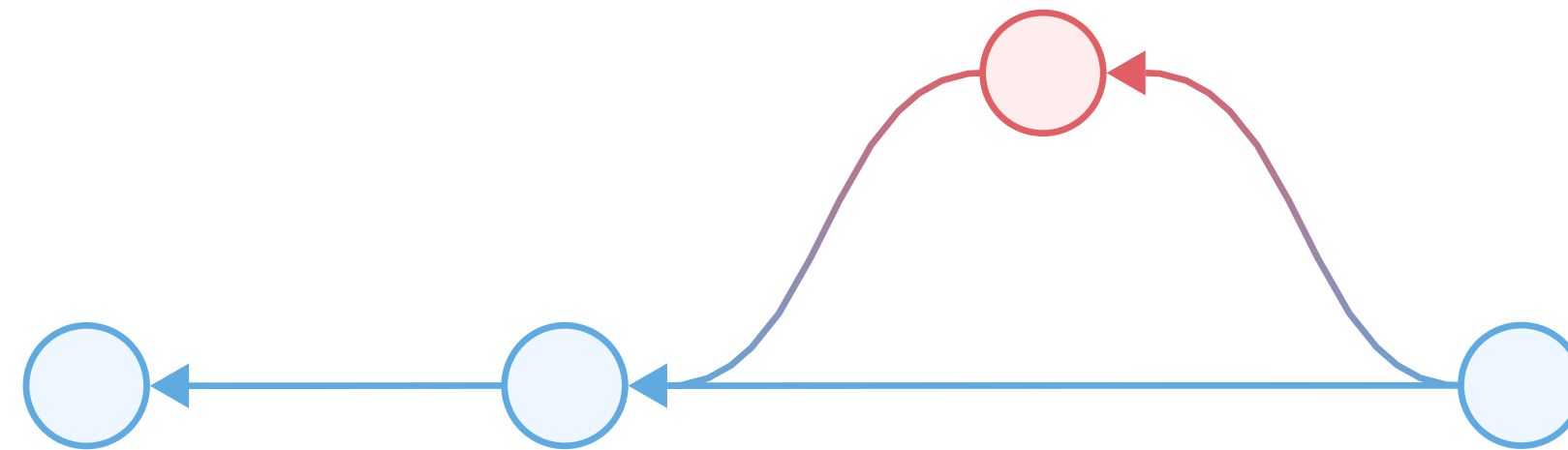
**Branches** allow a project to progress in different, independent directions. In practice, the graph is rarely a straight line.

Software developers often use branches for new features, which they subsequently merge into the main branch. This way, the main branch remains stable while developers are working on substantial changes.



## 📖 Terminology: Merges and rebases

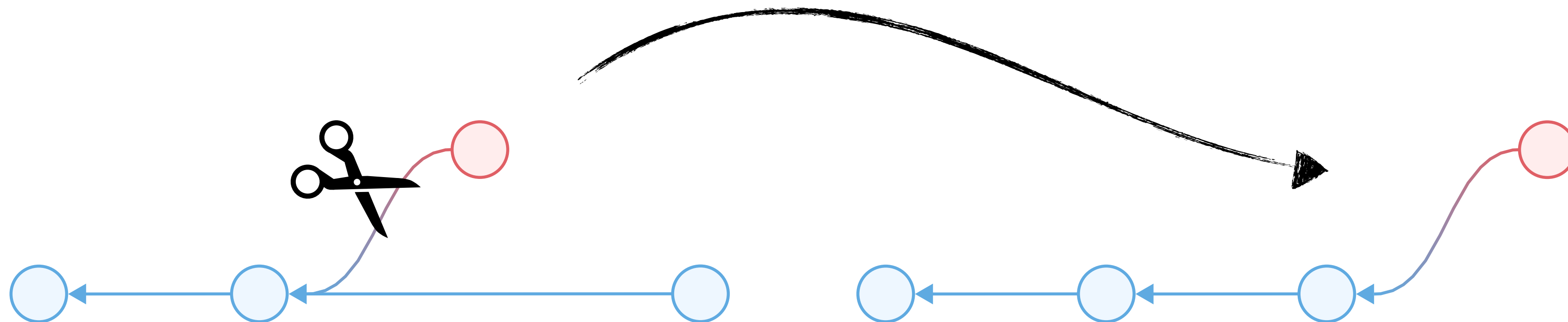
A **merge** combines commits from two or more branches into a new commit on a specific branch.



## 📖 Terminology: Merges and rebases

A **rebase** will change the parent of a commit. It also allows you to combine or modify commits.

Since it can edit the history of a project, rebasing is not recommended if you've already shared your commits with others (on a remote repository, for instance). We will not cover rebasing in the hands-on portion; we'll use merges when working with multiple branches.

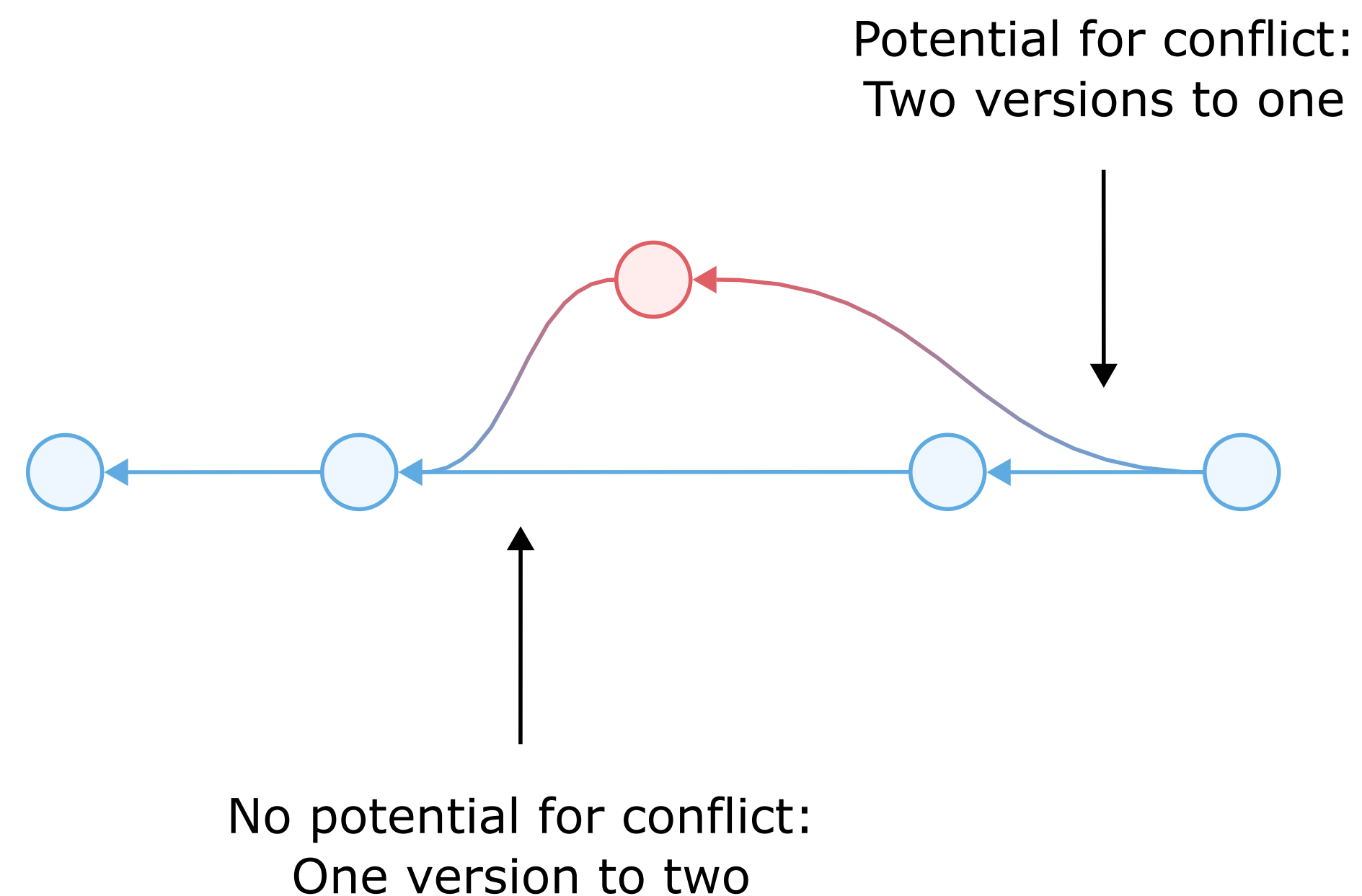


## Terminology: Conflicts

**Conflicts** can occur in several situations when you're using Git. They're a normal part of using Git and *not* an indication that you've done something wrong (the conflict is between versions, not between people!). They help prevent you from losing information or overwriting someone else's work.

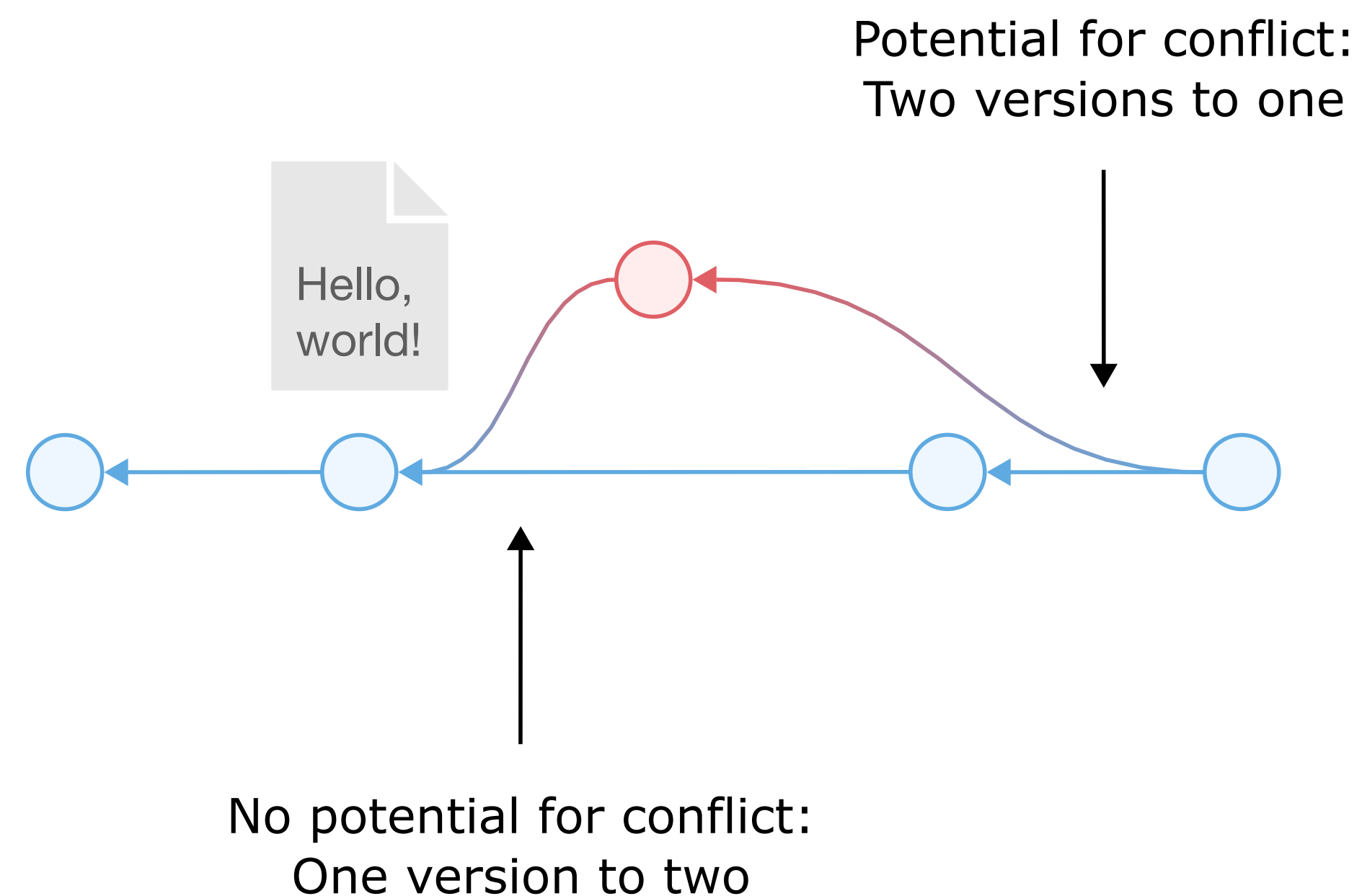
## Terminology: Merges ... and conflicts

When merging branches, you can encounter conflicts if the parent commits have modifications to the same parts of the project. Git has no way of knowing which (if any) is the authoritative or correct version.



## Terminology: Merges ... and conflicts

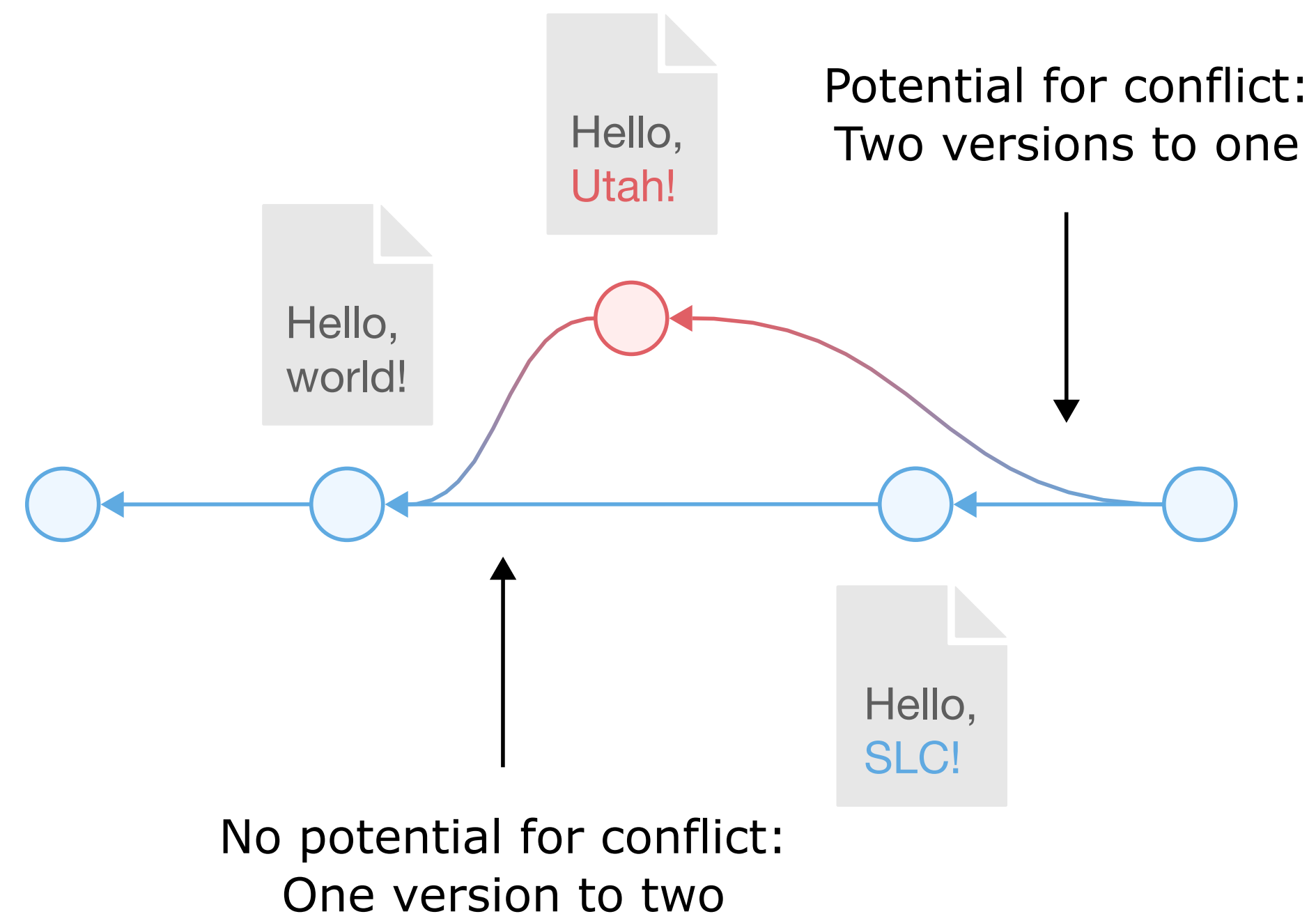
When merging branches, you can encounter conflicts if the parent commits have modifications to the same parts of the project. Git has no way of knowing which (if any) is the authoritative or correct version.





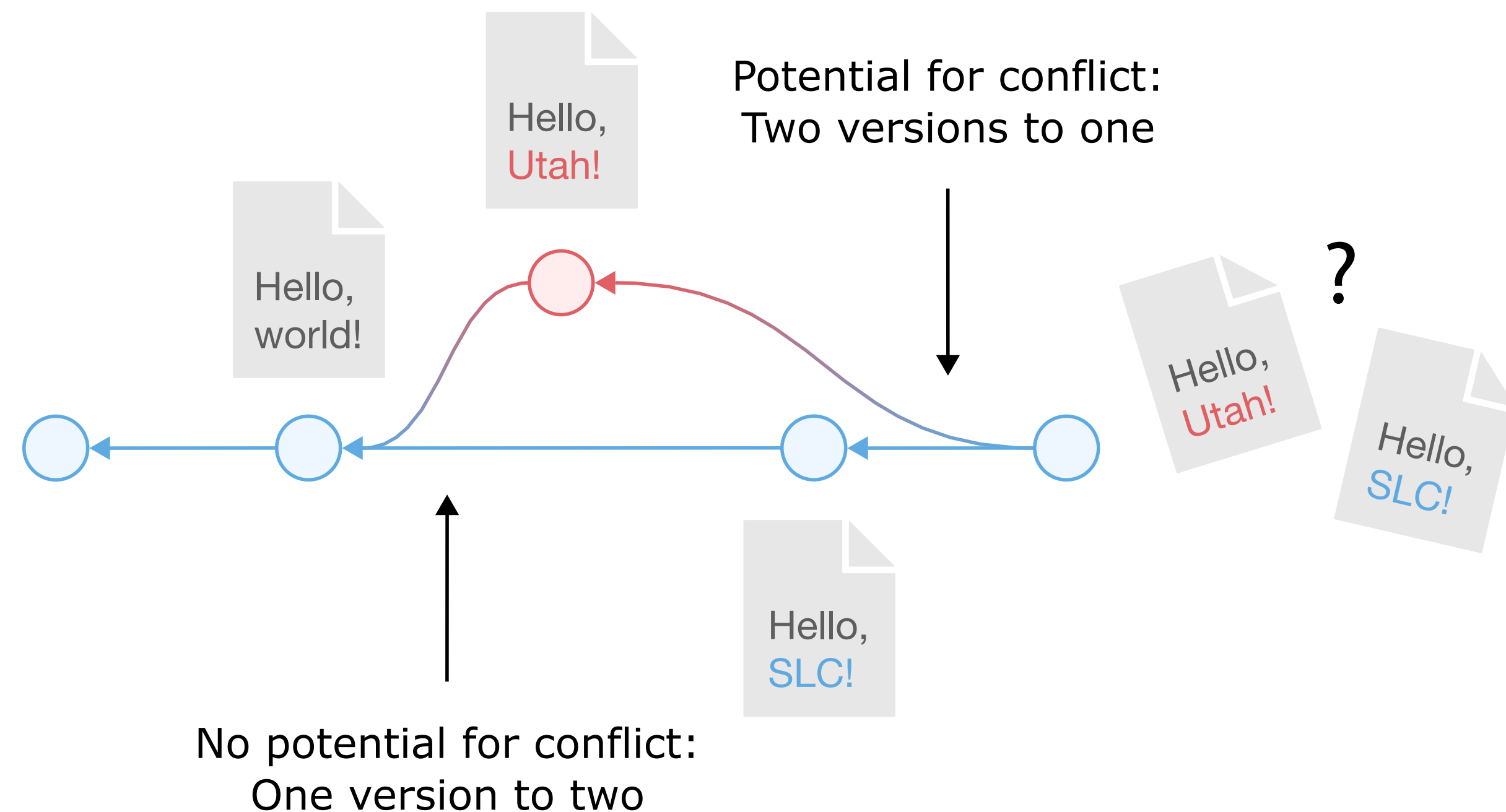
## Terminology: Merges ... and conflicts

When merging branches, you can encounter conflicts if the parent commits have modifications to the same parts of the project. Git has no way of knowing which (if any) is the authoritative or correct version.



## Terminology: Merges ... and conflicts

When merging branches, you can encounter conflicts if the parent commits have modifications to the same parts of the project. Git has no way of knowing which (if any) is the authoritative or correct version.

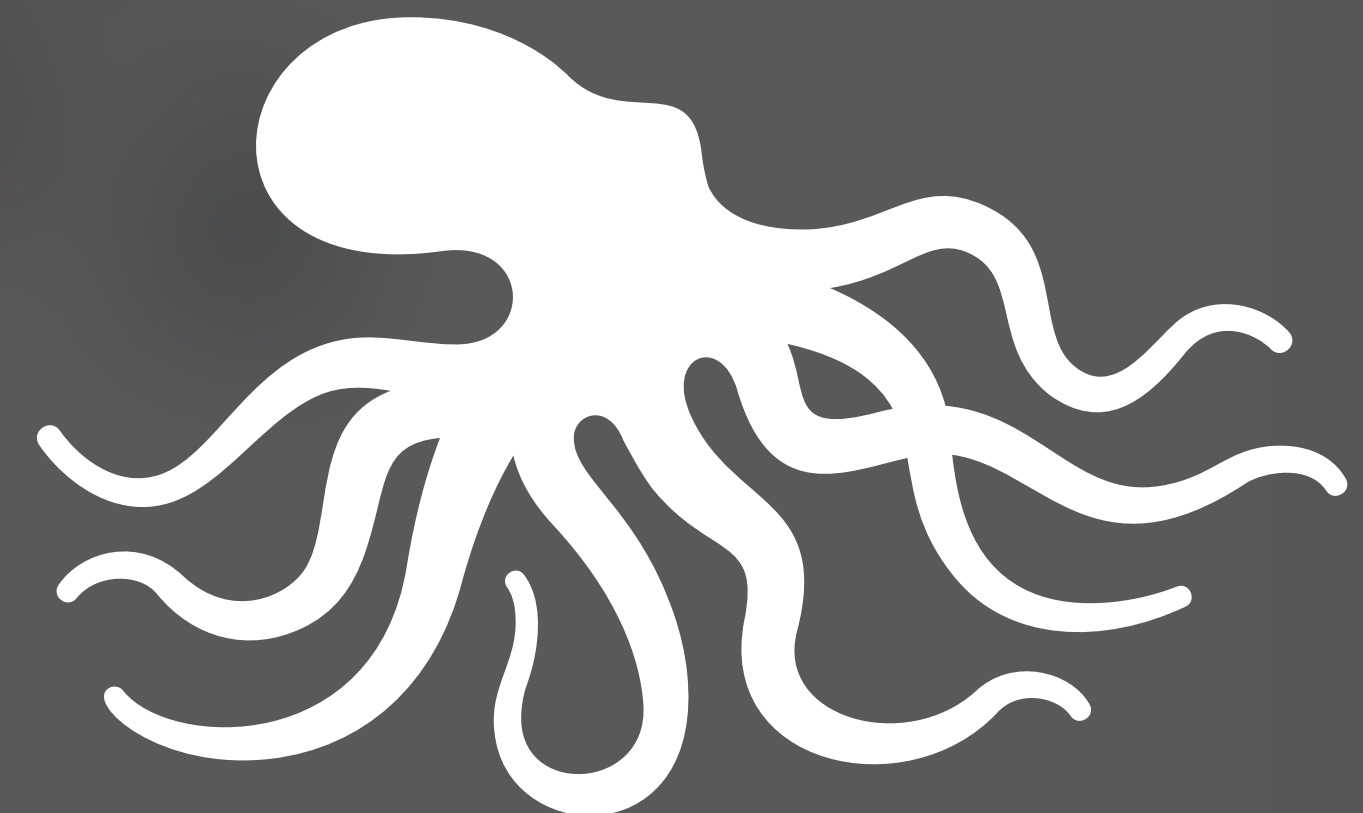
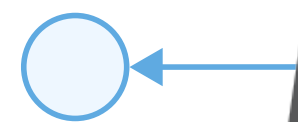


## 📖 Terminology: Merges

When merging branches  
parent commits have n  
project. Git has no way  
authoritative or correct

You can merge multiple branches  
at the same time.

This is sometimes called an “octopus merge.”  
Some Git users prefer to merge branches  
sequentially to reduce the number of  
conflicts that need to be addressed at a time.



## 📖 Terminology: Merges ... and conflicts

A merge with a conflict will result in text being added to the files with conflicts. Git will add *both versions* to the file on separate lines so you can pick one or write something new in their place. Once you've resolved all conflicts, you can create a new commit.

```
<<<<<<< HEAD
```

```
Hello, SLC!
```

```
=====
```


```
Hello, Utah!
```

```
>>>>>>> other_branch
```

## 📖 Terminology: Merges ... and conflicts

A merge with a conflict will result in text being added to the files with conflicts. Git will add *both versions* to the file on separate lines so you can pick one or write something new in their place. Once you've resolved all conflicts, you can create a new commit.

```
<<<<<<< HEAD           We'll talk about this later!  
Hello, SLC!  
=====  
Hello, Utah!  
>>>>>>> other_branch
```



# In practice: Branching and merging

To check which branch you're on, run `git branch`, which will list branches and show an asterisk `*` next to the current branch.

To create—and switch to—a new branch, use

```
git switch -c new_branch_name
```

To switch to an existing branch, use

```
git switch other_branch_name
```



# In practice: Branching and merging

To merge, switch to the branch you want to merge **into** (the destination branch) and run

```
git merge some_other_branch
```

New users sometimes forget which branch they should be on when merging. The branch you are on *when you perform the merge* is the branch to which the new commit will be added. Think of a merge as an extension of **git commit**.





# Hands-on: Branches and merges

1. Create a new branch (and switch to it)
2. Edit your files and commit your changes
3. Switch back to your other branch (how do you get a list of the names of branches?)
4. Perform a merge to incorporate changes you made in your new branch
5. Can you cause (and resolve) a merge conflict?



## Terminology: References and HEAD

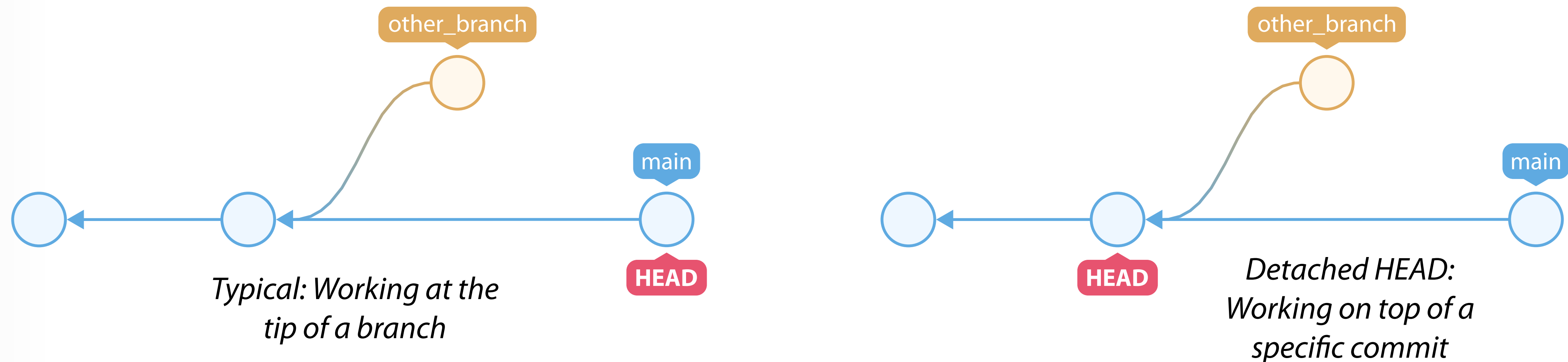
We've been using the names of branches in commands. These are **references** to specific commits. Branch references refer to the tip (the most recent commit) of a branch. This may also be called the head (not to be confused with the HEAD).

We also saw **HEAD** earlier. HEAD refers to the current commit: the one we're looking at right now. Usually, HEAD is a head (the tip of a branch), but we can move it anywhere we want to look at or work on files *as they were at that commit*. This is called a "detached HEAD."

## 📖 Terminology: References and HEAD

We've been using the names of branches in commands. These are **references** to specific commits. Branch references refer to the tip (the most recent commit) of a branch. This may also be called the head (not to be confused with the HEAD).

We also saw **HEAD** earlier. HEAD refers to the current commit: the one we're looking at right now. Usually, HEAD is a head (the tip of a branch), but we can move it anywhere we want to look at or work on files *as they were at that commit*. This is called a "detached HEAD."



## 📖 Terminology: References and HEAD

Remember that, according to Chacon and Straub (*Pro Git*), “Git basically takes a picture of what all your files look like” when you make a commit.

If your commits are a series of pictures, HEAD is the one you’re looking at (working on) *right now*.

*Image credit: Girl with red hat on Unsplash  
Unsplash License*



## Terminology: Reverting and resetting

References are an important concept as we start discussing **reverting and resetting**. While I mentioned that Git is not the same as having an undo button, sometimes you *do* want to undo changes. ↶

There are a few different strategies for this. Before we get into the different options, it's important to note that *nondestructive operations are always preferred if you are working with other people (if you've shared your repository)*. In other words, actually *deleting* information should be done sparingly—preferably *before* you update a remote repository or send your files to someone.

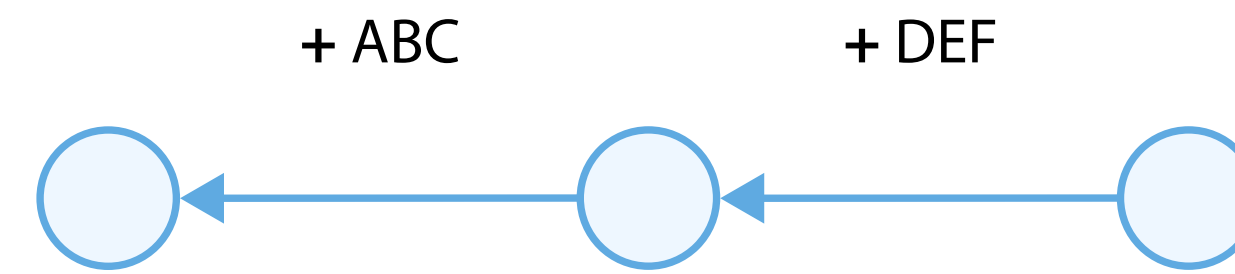


## Terminology: Reverting

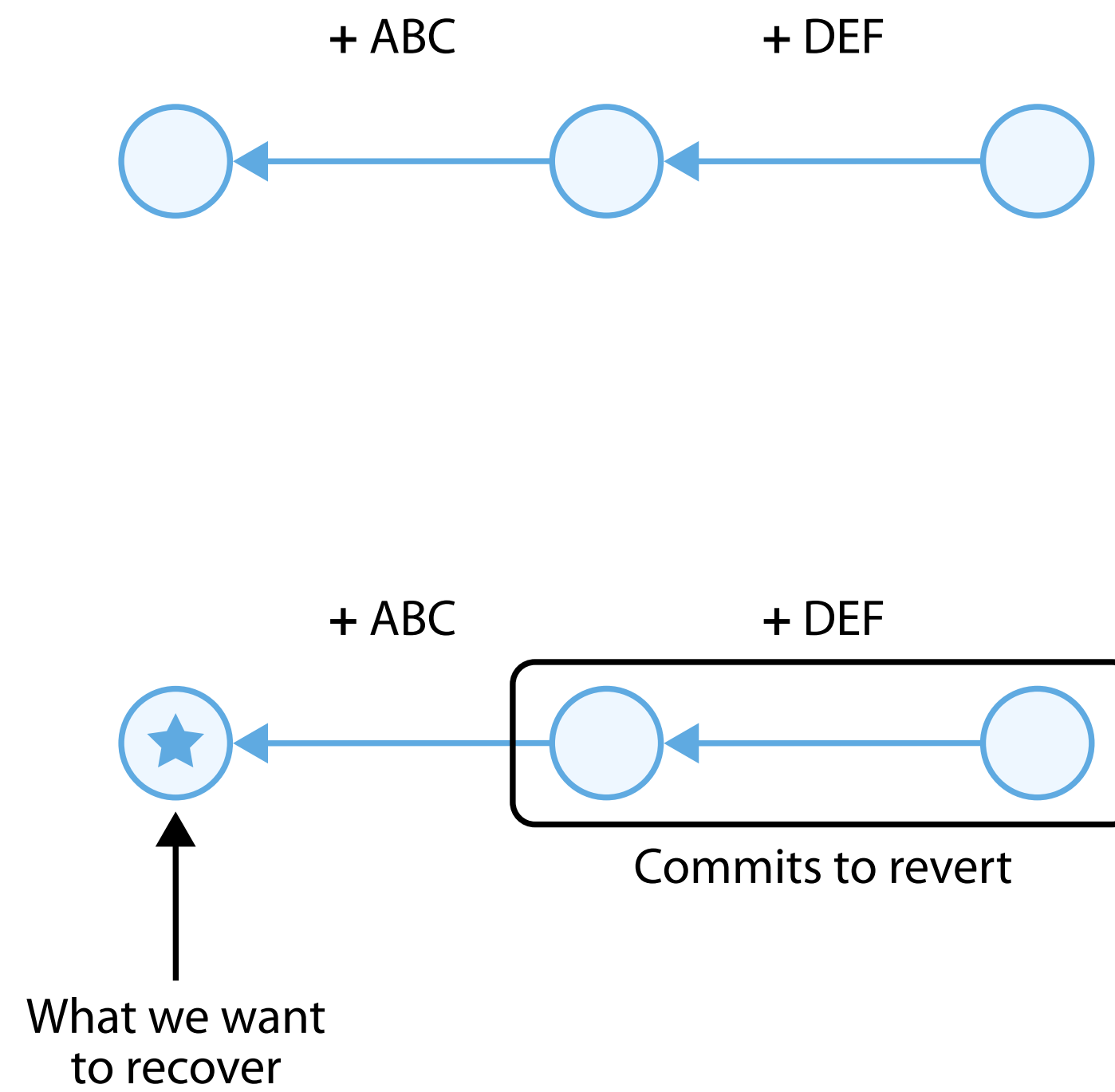
**Reverting** is a nondestructive option to undo changes. Reverting always creates new commits; it does not remove anything from the project history.

There are a few quirks that you should be aware of before you start reverting. Please refer to the handout for more information.

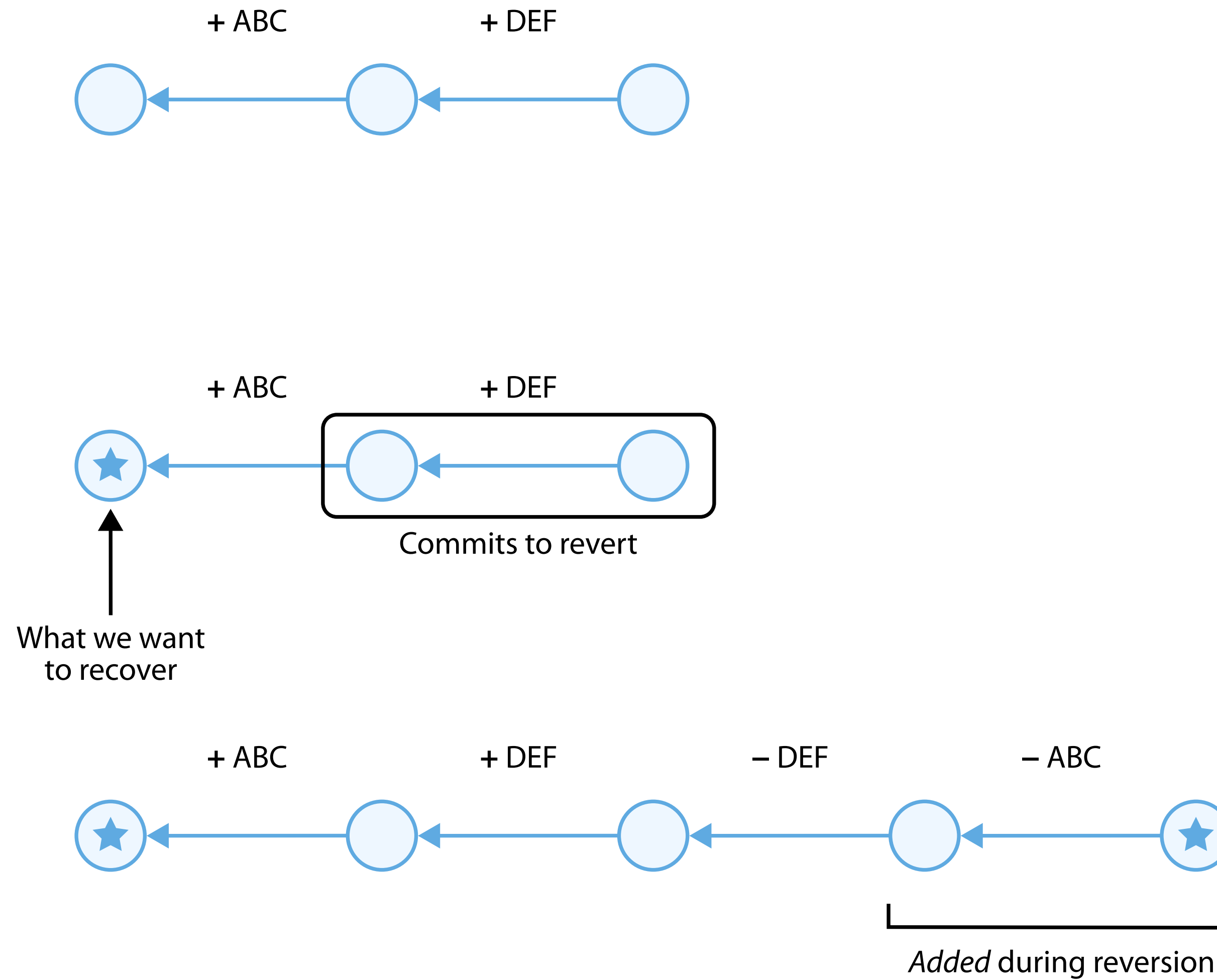
## Terminology: Reverting



## Terminology: Reverting



## Terminology: Reverting



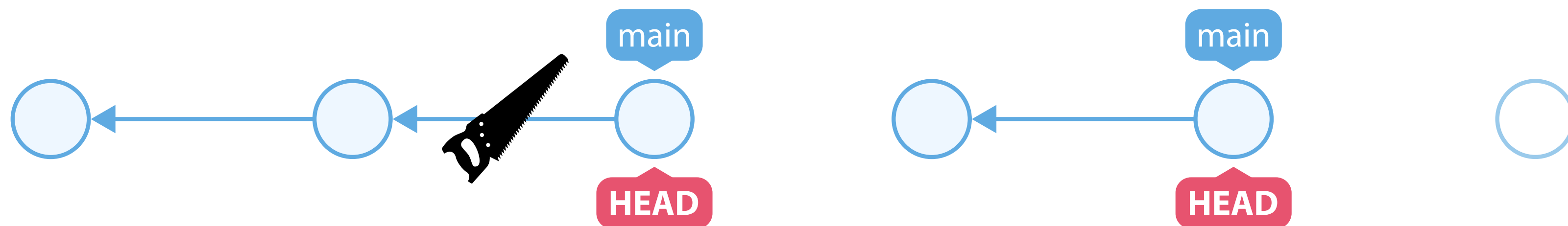


## 📖 Terminology: Resetting

**Resetting** can be a destructive option to undo changes. It works by moving the branch reference (and HEAD).

This can “orphan” commits (it doesn't actually delete them). If you accidentally reset, you may be able to recover your commits. They will, however, be deleted when Git performs garbage collection.

There are a few quirks that you should be aware of before you start resetting. Please refer to the handout for more information.



# In practice: Checking out, reverting, and resetting

To work “on top of” a specific commit, use

`git checkout some-commit`

To revert, use

`git revert some-commit`

See handout for more information; this only operates on one commit unless you specify a range

To reset, use

`git reset some-commit`

See handout for more information; this can be `--hard`, `--mixed` (default), or `--soft`





# Hands-on: Branches and merges

1. Create a new commit; run a `git diff` on the last two commits
2. `revert` your new commit; run a `git diff` on the last two commits
3. `checkout` a previous commit (how do you see previous commits?)
4. Make a new branch (and switch to it)
5. Create a new commit, then run `git log --graph --all`

## Terminology: Remote repositories

**Remote repositories** are Git repositories that are not local to your computer. They're often hosted on platforms like GitHub or GitLab.

You can interact with remote repositories with

- **git clone** to copy a remote repository to your own computer
- **git remote** to configure a remote repository's information locally
- **git fetch** to get changes from a remote repository
- **git pull** to get changes from a remote repository and merge them into your local repository
- **git push** to upload changes from your local repository to a remote repository

See the handout for more information.



## Terminology: Remote repositories

Remote repository hosts often add features that are not a part of Git itself. These include such features as

## Terminology: Remote repositories

Remote repository hosts often add features that are not a part of Git itself. These include such features as

- **Issues**, which allow users to ask questions or report problems

## Terminology: Remote repositories

Remote repository hosts often add features that are not a part of Git itself. These include such features as

- **Issues**, which allow users to ask questions or report problems
- **Pull requests**, which allow users to suggest changes
  - These are important when working with others; generally, only a limited set of users is allowed to write to the repository itself
  - Other users first make a copy, then make changes on their copy, then suggest the changes to the maintainers of the original repository

## Terminology: Remote repositories

Remote repository hosts often add features that are not a part of Git itself. These include such features as

- **Issues**, which allow users to ask questions or report problems
- **Pull requests**, which allow users to suggest changes
  - These are important when working with others; generally, only a limited set of users is allowed to write to the repository itself
  - Other users first make a copy, then make changes on their copy, then suggest the changes to the maintainers of the original repository
- **Continuous integration and continuous delivery** (build and test suites that run every time the repository is updated)

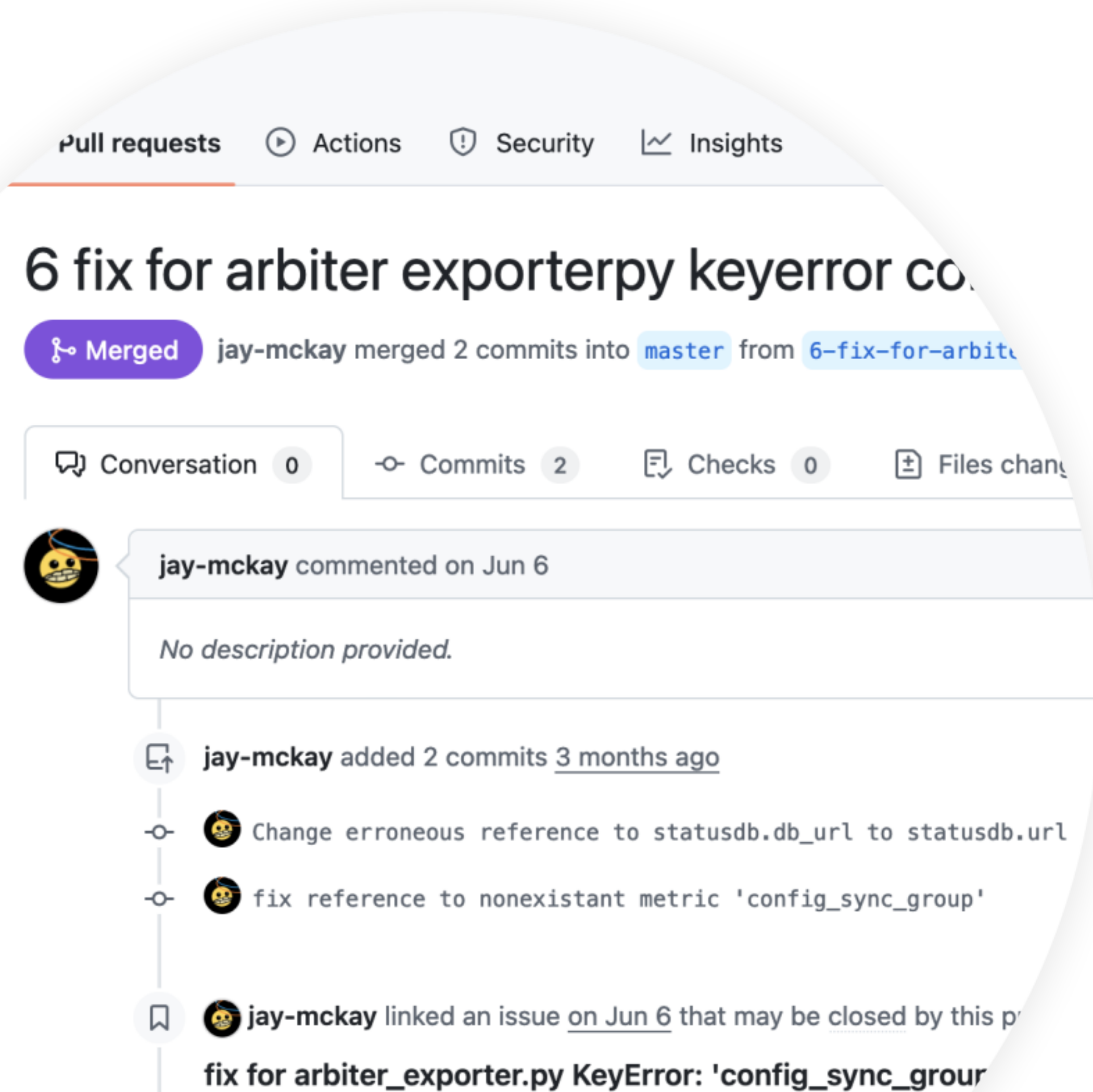


## Terminology: Remote repositories

Remote repository hosts often add features that are not a part of Git itself. These include such features as

- **Issues**, which allow users to ask questions or report problems
- **Pull requests**, which allow users to suggest changes
  - These are important when working with others; generally, only a limited set of users is allowed to write to the repository itself
  - Other users first make a copy, then make changes on their copy, then suggest the changes to the maintainers of the original repository
- **Continuous integration and continuous delivery** (build and test suites that run every time the repository is updated)
- **Machine learning-driven security reviews** (a recent development)

## 📖 Terminology: Remote repositories



The screenshot shows a GitHub pull request interface. At the top, there are navigation tabs: "Pull requests" (selected), "Actions", "Security", and "Insights". The main title of the pull request is "6 fix for arbiter exporter.py keyerror co...". Below the title, a purple "Merged" badge is visible, followed by the text "jay-mckay merged 2 commits into master from 6-fix-for-arbit...".

Below the merge information, there are statistics: "Conversation 0", "Commits 2", "Checks 0", and "Files changed".

The main content area shows a comment from "jay-mckay" dated "Jun 6". The comment text is "No description provided." Below the comment, there is a commit history section. The first entry is "jay-mckay added 2 commits 3 months ago". This is followed by two commit messages, each preceded by a commit icon:

- Change erroneous reference to statusdb.db\_url to statusdb.url
- fix reference to nonexistant metric 'config\_sync\_group'

At the bottom, there is a link icon followed by the text "jay-mckay linked an issue on Jun 6 that may be closed by this pr". The issue title is partially visible as "fix for arbiter\_exporter.py KeyError: 'config\_sync\_grou...".


## 📖 Terminology: Remote repositories

Pull requests Actions Security Insights


### 6 fix for arbiter exporter.py keyerror co





Merged jay-mckay merged 2 commits into master from 6-fix-for-arbit



Conversation 0 Commits 2 Checks 0 Files

 jay-mckay commented on Jun 6

*No description provided.*

 jay-mckay added 2 commits 3 months ago

-   Change erroneous reference to statusdb.db\_ur
-   fix reference to nonexistant metric 'config\_


  jay-mckay linked an issue on Jun 6 that may be c

fix for arbiter\_exporter.py KeyError: 'config\_sy

Pull requests Actions Security Insights


### Update to include compatibility with

Open snowbird294 opened this issue on Feb 8, 2023 · 9 comments

 snowbird294 commented on Feb 8, 2023

RHEL 9 was released after the last published update of arbiter2. A number of 9. Are there plans to update arbiter to match?

The first error I'm running into is "ERROR: processes.memsw = TRUE" isn't ava about "cgroup hierarchy doesn't exist." I can see at least part of the cgroups hi including memory inside cgroup.controllers. I believe this indicates that cgroup issue with Arbiter.

 1

## Terminology: Remote repositories

The Center for High Performance Computing has a GitLab instance ([gitlab.chpc.utah.edu](https://gitlab.chpc.utah.edu)) that you can use for your projects. It uses your University of Utah credentials. You need to be on the university's network to use it, so you will need to use the university's VPN from anywhere off-campus.

# You now know enough about Git to use it for your projects!

We'll continue with a few more topics, but we will likely not have time for hands-on exercises. Please refer to the handout for examples of commands.





## Terminology: Stash

You can *stash* changes (your working tree and index) in the **stash**. This is useful to clear out changes you don't need *right now* but may need at some point. You might use this, for example, when you need a “clean” working tree or index.

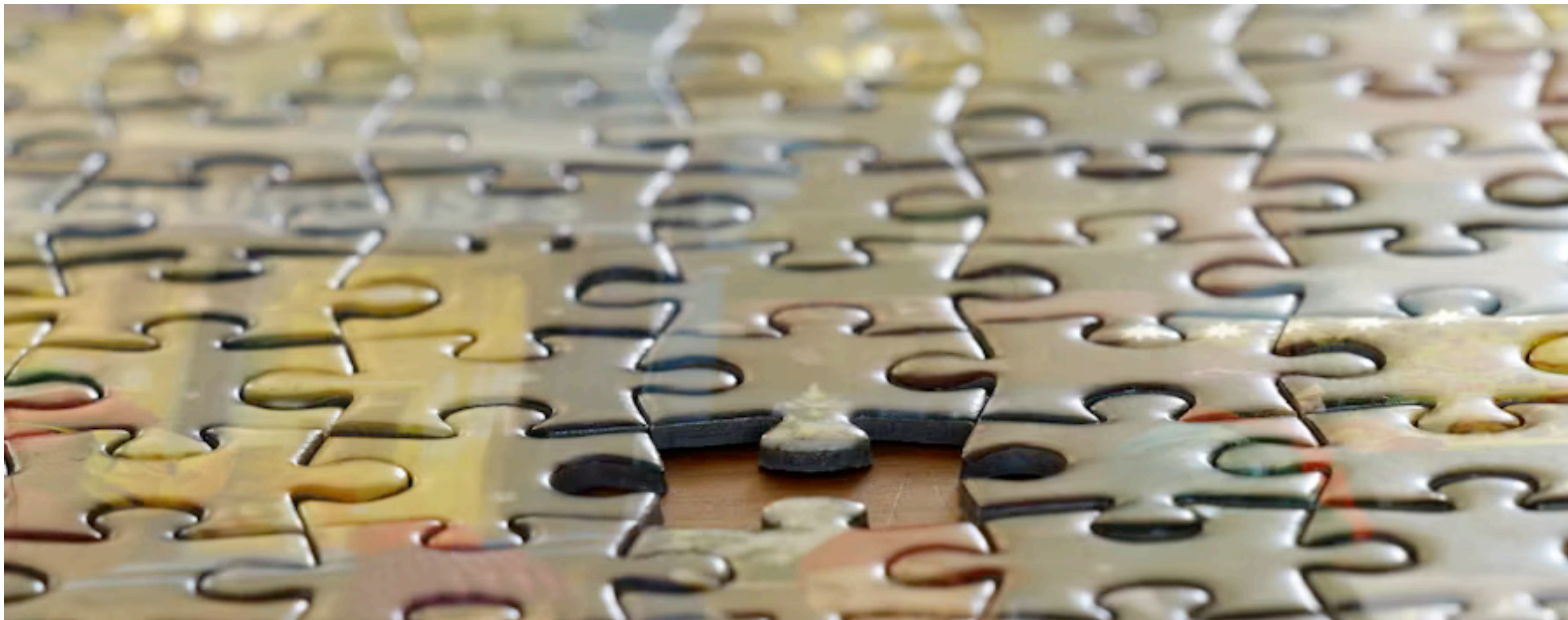
You could also make a commit on a new branch, then come back to that branch later if you wanted to incorporate your changes into a different branch.

Examples of using **git stash** are provided in the handout.

## 📖 Terminology: Submodules

A **submodule** is another Git repository that you include as part of your repository. This is the preferred way to refer to other projects, as it allows you to reference specific commits in another project (without including the contents of the other repository in your own).

*Image credit: Pierre Bamin on Unsplash  
Unsplash License*



## Terminology: Large File Storage (LFS)

**Git LFS** is an extension used for versioning large files. It uses “text pointers inside Git,” while the actual files are stored elsewhere. It’s used to keep repositories (relatively) small.



## Terminology: Large File Storage (LFS)

**Git LFS** is an extension used for versioning large files. It uses “text pointers inside Git,” while the actual files are stored elsewhere. It’s used to keep repositories (relatively) small.

One reason to do this is so that users don’t need to download every past version of large files when they **clone** your project. (If you have a 20 GB file with 5 revisions, that’d be 100 GB to download—and most of your users probably wouldn’t care about the 80 GB of old content that has since been updated.)

## Terminology: Large File Storage (LFS)

**Git LFS** is an extension used for versioning large files. It uses “text pointers inside Git,” while the actual files are stored elsewhere. It’s used to keep repositories (relatively) small.

One reason to do this is so that users don’t need to download every past version of large files when they **clone** your project. (If you have a 20 GB file with 5 revisions, that’d be 100 GB to download—and most of your users probably wouldn’t care about the 80 GB of old content that has since been updated.)

Note that hosts like GitHub will *reject* larger files (as of Fall 2024, GitHub “blocks files larger than 100 MB” and warns about files larger than 50 MB.)



## Terminology: Tags

**Tags** allow you to give a name to a commit. They are commonly used to annotate specific commits with version numbers (e.g., “v1.2”).

*Image credit: Kelsy Gagnebin on Unsplash  
Unsplash License*



**There are a few different files that are commonly added to repositories by developers and researchers.**



## **.gitignore**

A **.gitignore** file allows you to designate files as “intentionally untracked.”

This is useful when you don't want files to end up in your repository, and it allows you to continue using flags like **--all** without worrying about things like intermediate files (e.g., \*.log and \*.aux with TeX documents, object files and binaries, user-specific configurations that others would not be interested in).

You can also use a different file by configuring `core.excludesFile`.

## .gitattributes

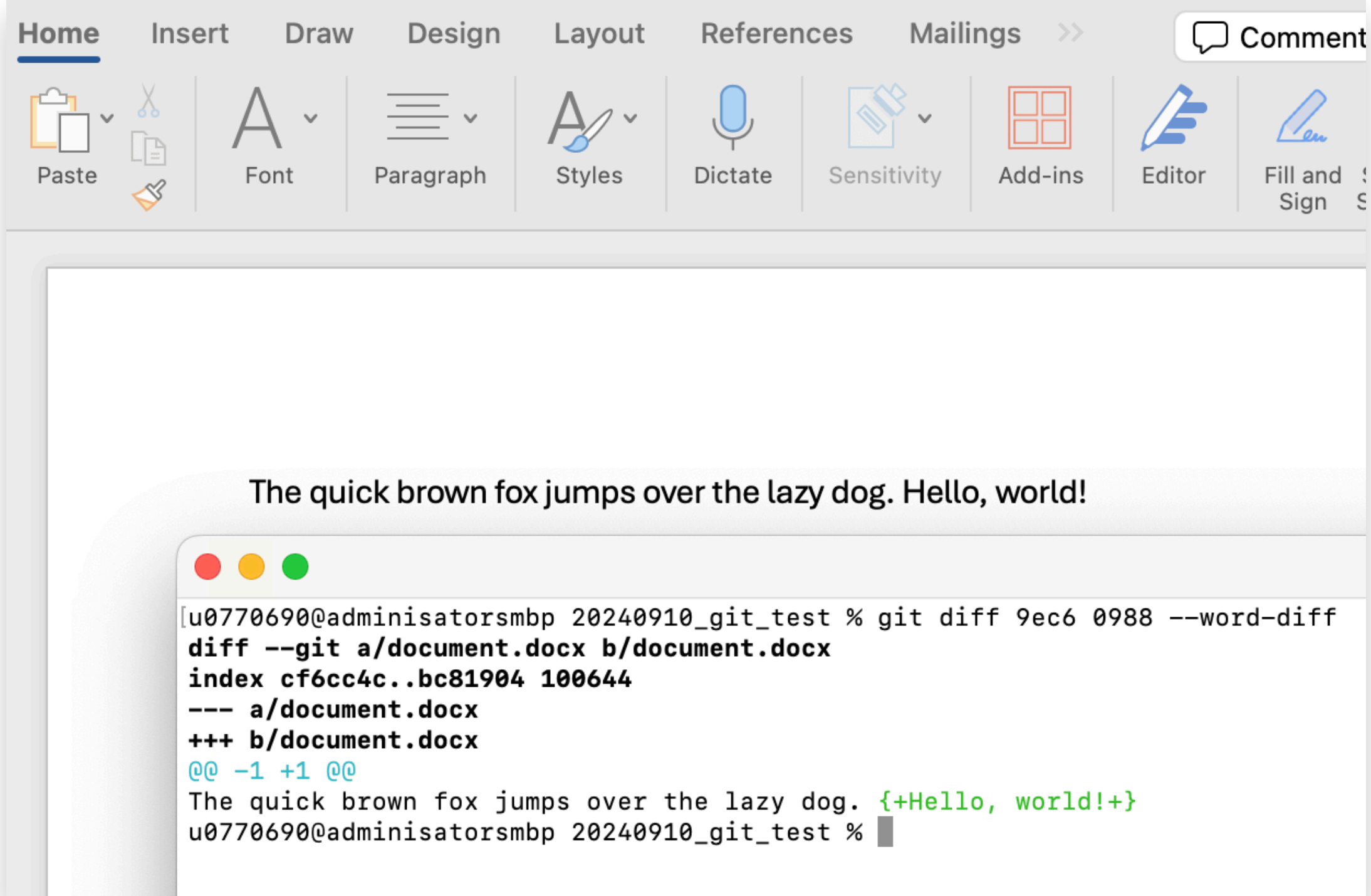
A **.gitattributes** file allows you to specify the attributes of specific files. This lets you, for example, identify certain files as binary files and change **diff** behavior.

*In .git/config:*

```
[diff "pandoc"]
  textconv = pandoc --to=markdown
```

*In .gitattributes:*

```
*.docx binary diff=pandoc
```



```
Home Insert Draw Design Layout References Mailings >> Comment
Paste Font Paragraph Styles Dictate Sensitivity Add-ins Editor Fill and Sign
The quick brown fox jumps over the lazy dog. Hello, world!
[u0770690@administratorsmbp 20240910_git_test % git diff 9ec6 0988 --word-diff
diff --git a/document.docx b/document.docx
index cf6cc4c..bc81904 100644
--- a/document.docx
+++ b/document.docx
@@ -1 +1 @@
The quick brown fox jumps over the lazy dog. {+Hello, world!+}
u0770690@administratorsmbp 20240910_git_test %
```

## **README**

A **README** file tells potential users and contributors about your project. The README is often written in Markdown, which allows you to add headings, links, tables, images, and other features. This is what's displayed on project pages on GitHub and similar sites.

We recommend including a README in every project!

 **LICENSE**

A **LICENSE** file tells users the conditions under which they can use, redistribute, and even commercialize your project contents. Potential users or contributors may avoid your project if the license is unclear.

We recommend including a LICENSE in every project!



**We've only just scratched the surface of Git today. There are many other, more advanced topics that we don't have time to cover in detail.**

The material we've covered today may be sufficient for your workflow, though, and you should now be able to understand the more advanced topics as you encounter them. We've made a lot of progress!



## Do you have any questions?

If we don't have time to answer your question, or if you think of any questions after the presentation, please reach out to us! [helpdesk@chpc.utah.edu](mailto:helpdesk@chpc.utah.edu)

