

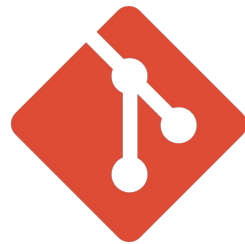
Version Control

ITI – Day 1 & 2



By Mohamed ElSabagh & Mahmoud Meshrif

Content



1

Introduction to VCS

What it is, use cases and advantages

2

Centralized VCS

The structure of the centralized VCS

3

Distributed VCS

The structure of the distributed VCS

4

Git & GitHub

History, advantages and hands-on

Content



5

**Branching &
Rebasing**

6

Pull Request

7

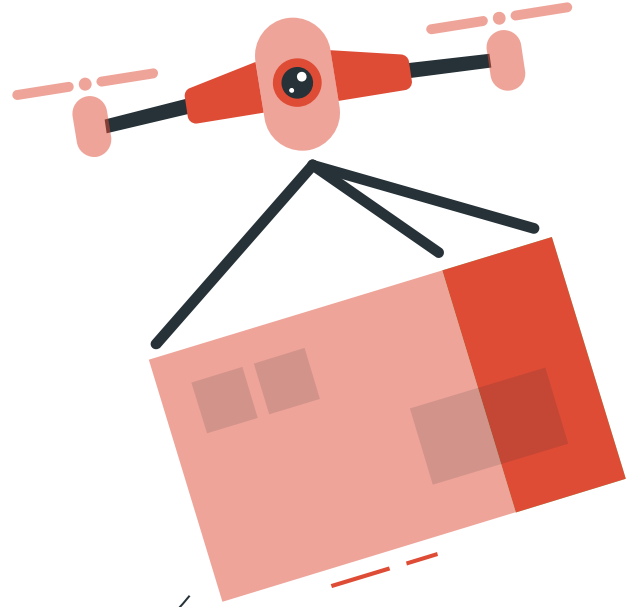
**Tagging &
Versioning**

8

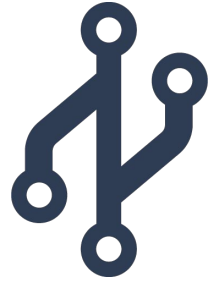
**Ignoring
Files**

Introduction To VCS

What it is, use cases
and advantages



What is version control?



Version control, also known as **source control**, is the practice of tracking and managing changes to software code.

Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake.

Why use Version Control?

1

Helps teams collaborate around the world



2

Accelerates product delivery



3

You can version and backup your code



4

Keeping Track of All the Modifications Made to the Code



5

Working on a new features without affecting the working code



Version Control Terminologies



Repository
Or Repo

Server

Client

Working
Copy

Master/Main

Match The VCS Terminologies

Repository (Repo)

1

Server

2

Client

3

Working Copy

4

Master/Main

5

A

The computer connecting to the repo

B

Local directory of files, where you make changes

C

The database storing the files

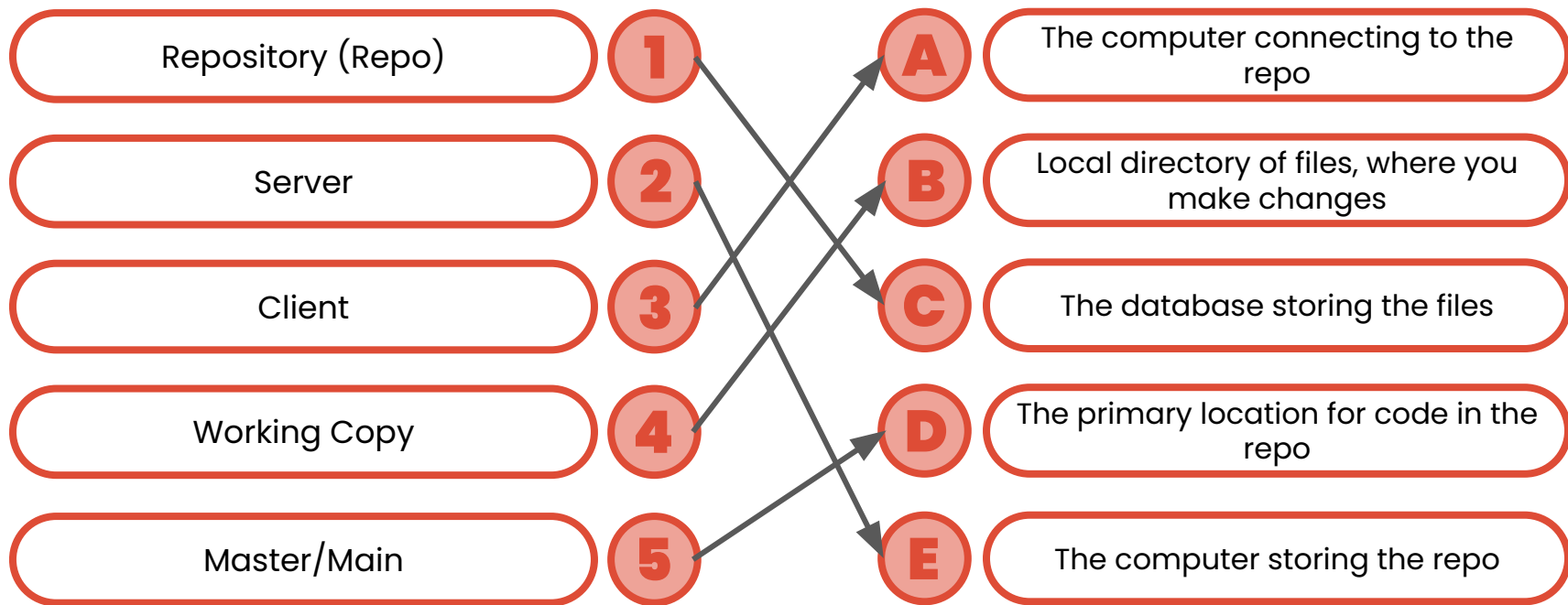
D

The primary location for code in the repo

E

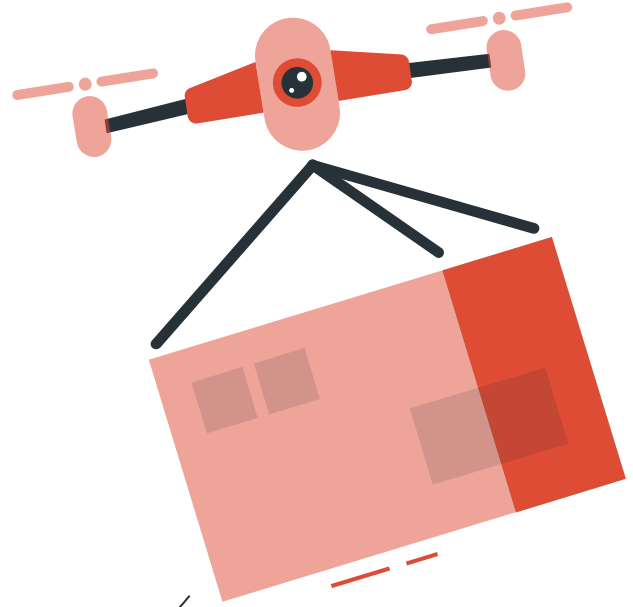
The computer storing the repo

The VCS Terminologies Solution

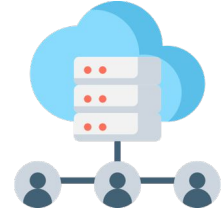


Centralized VCS

The structure of
the centralized VCS



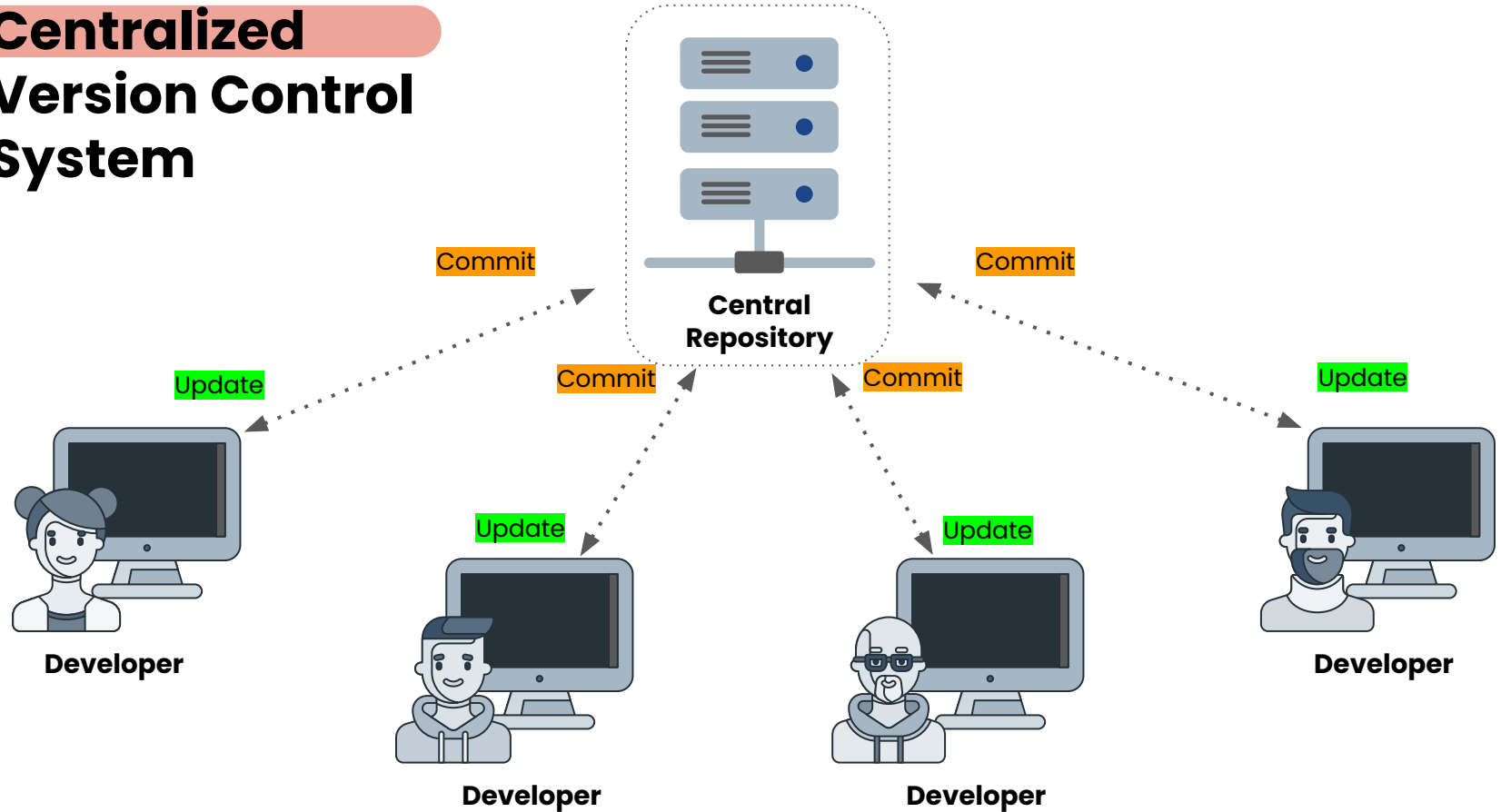
Centralized VCS



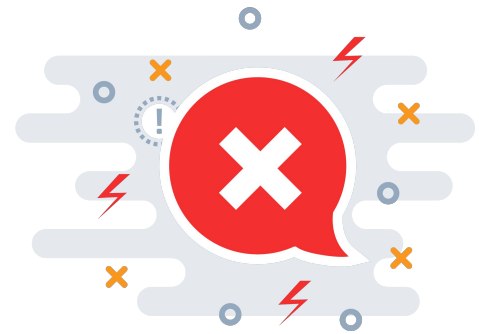
Centralized version control systems are based on the idea that there is a single “central” copy of your project on a server, and programmers will “commit” their changes to this central copy.

“Committing” a change simply means recording the change in the central system. Other programmers can then see this change. They can also pull down the change, and the version control tool will automatically update the contents of any files that were changed.

Centralized Version Control System



Disadvantages



- If the main server **goes down**, developers can't save versioned changes.
- Need **internet connection** to commit the changes.
- **Unsolicited** changes might ruin development.
- If the central database is corrupted, the entire history could be lost.

Centralized VCS Examples



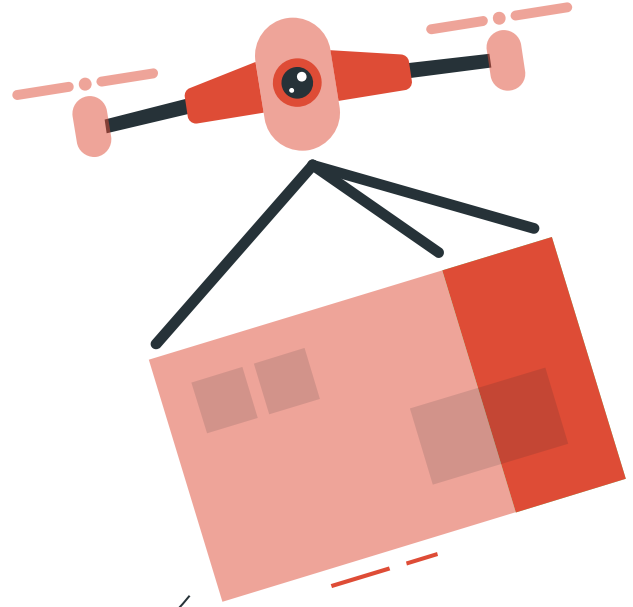
Concurrent
Version System



Subversion

Distributed VCS

The structure of
the distributed VCS



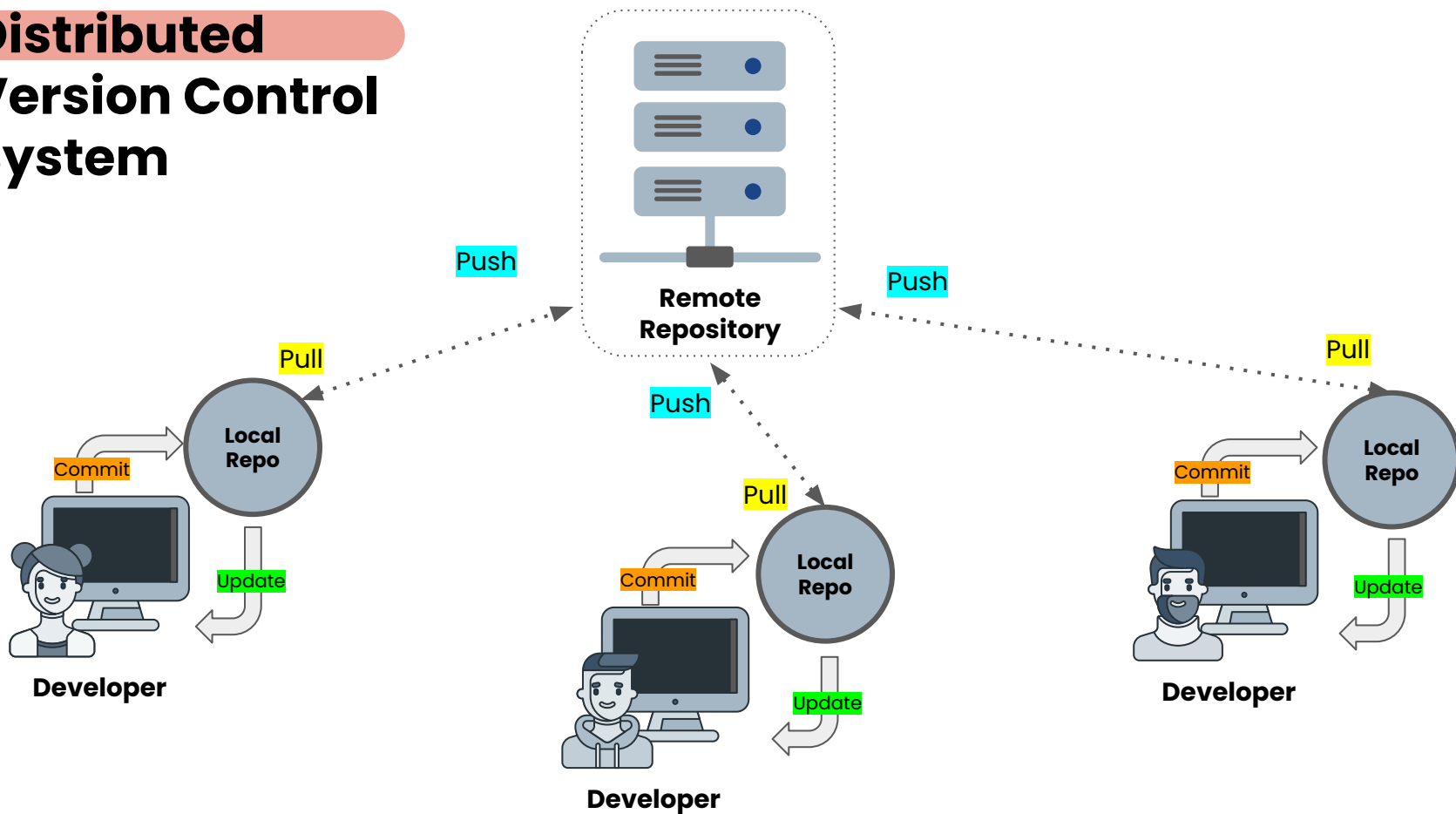
Distributed VCS



These systems do **not necessarily** rely on a central server to store all the versions of a project's files. Instead, every developer "**clones**" a copy of a repository and has the full history of the project on their own hard drive.

The act of getting new changes from a repository is usually called "**pulling**", and the act of moving your own changes to a repository is called "**pushing**". In both cases, you move changesets (changes to files groups as coherent wholes), not single-file diffs.

Distributed Version Control System



Advantages



- **Performing actions** other than pushing and pulling changesets is extremely fast because the tool only needs to access the hard drive, not a remote server.
- Committing new changesets can be **done locally** without anyone else seeing them. Once you have a group of changesets ready, you can push all of them at once.

Advantages – Cont'd



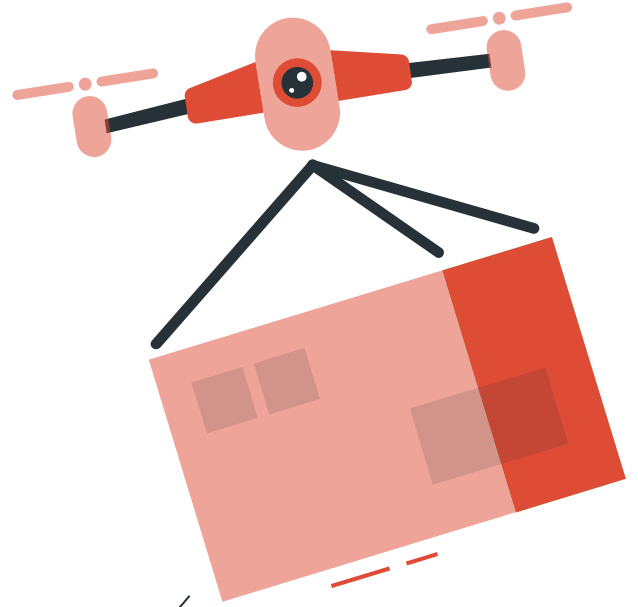
- Everything but pushing and pulling can be done without an internet connection. So you can work on a plane, and you won't be forced to commit several bugfixes as one big changeset.
- Since each programmer has a full copy of the project repository, they can share changes with one or two other people at a time if they want to get some feedback before showing the changes to everyone.

Distributed VCS Examples



Git & GitHub

History, advantages
and hands-on

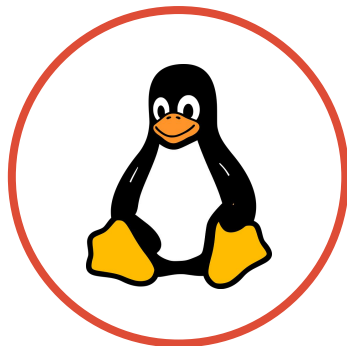


Git History



2002

Linux kernel project
began using
BitKeeper



2005

Linux kernel project
stopped using
BitKeeper



2005

Linus Torvalds
started working on a
new DVCS called Git

Simple design

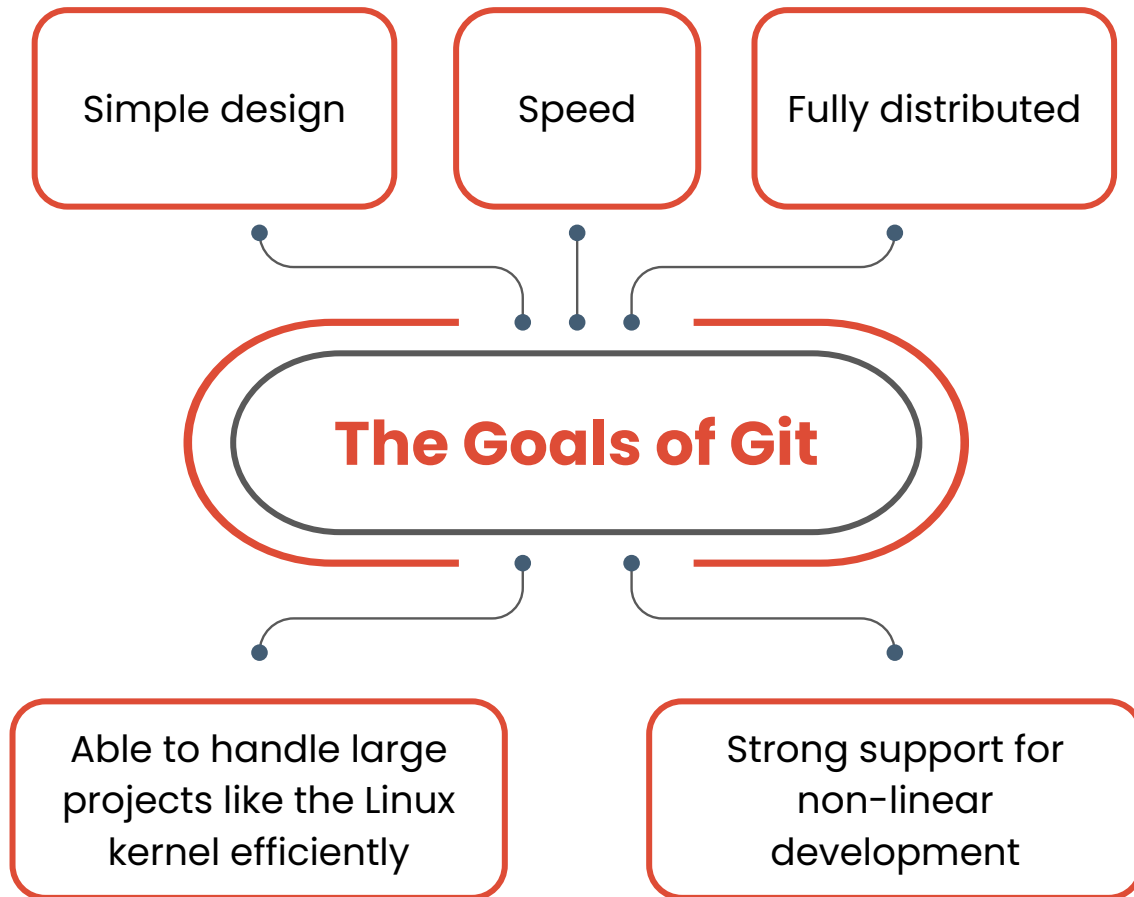
Speed

Fully distributed

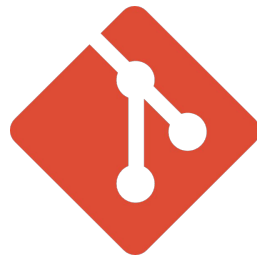
The Goals of Git

Able to handle large
projects like the Linux
kernel efficiently

Strong support for
non-linear
development

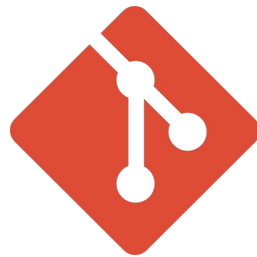


Every Operation is Local



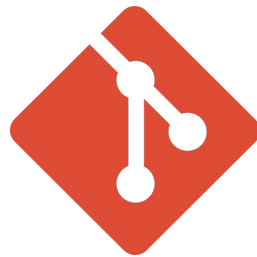
- Most operations in Git **only need local files** and resources to operate, generally no information needed from another computer on your network.
- **For example:** to browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you, it simply reads it directly from your local database.

Git Integrity



- The Git Version Control System uses **SHA-1 checksums** on the contents of all change commits. In fact, the checksum is used as commit identifier and commonly referred to as "the SHA". Git's checksums include metadata about the commit including the author, date, and the previous commit's SHA.
- Git assures the integrity of the data being stored by using checksums as identifiers. If someone were to try to alter a commit or its meta data, it would change the SHA used to identify it. It would become a different commit.

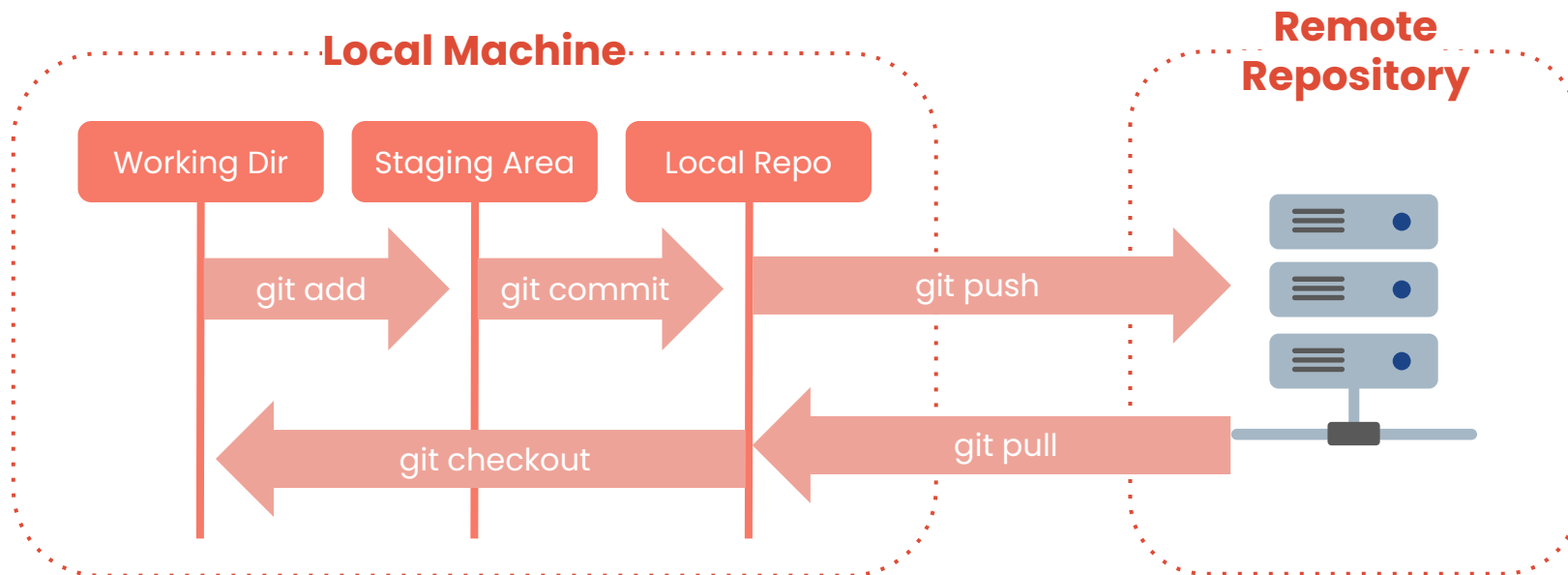
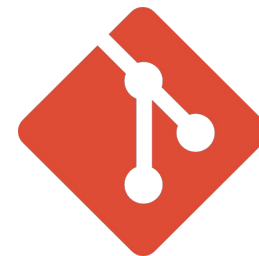
The Three Git States



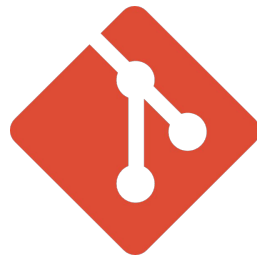
Git has three main states that your files can reside in:

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

The Three Git States



First Time Git Setup



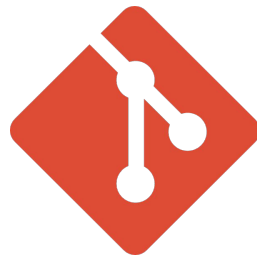
- The first thing you should do when you install Git is to set your user name and email address.
- This is important because every Git commit uses this information, and it's immutably baked into commits you start creating.

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

- You can also set your default editor and colorize the output:

```
git config --global core.editor "code --wait"  
git config --global color.ui true
```

Git SSH Keys



SSH keys come in pairs, **public key** that gets shared with services like GitHub and a **private key** that is stored only on your computer. If the keys match, you're granted access.

- **Generate a new SSH key pairs**
`ssh-keygen -t ed25519 -C "xxx@gmail.com"`
- **Copy the public key to your GitHub account**
`cat ~/.ssh/id_ed25519.pub`

Starting a Repo



- `mkdir python_project`

Make a directory

- `cd python_project`

Change directory to the above directory

- `git init`

Initialize an empty Git repository

- `ls -a`

List all the files & dir and the hidden files & dir

Create a New File

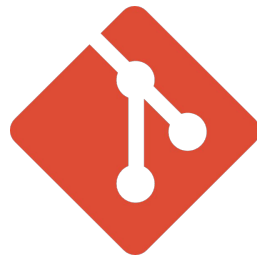


- `touch script.py`
Create a new file called `script.py`
- `git status`

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        script.py

nothing added to commit but untracked files present (use "git add" to track)
→ python_project git:(master) x
```

Add to Staging Area

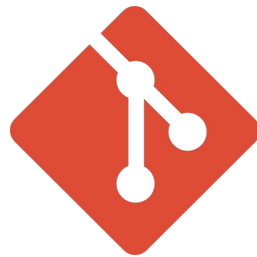


- `git add script.py`
Staging the script.py file
- `git status`

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
      new file:   script.py

→ python_project git:(master) x
```


Commit changes



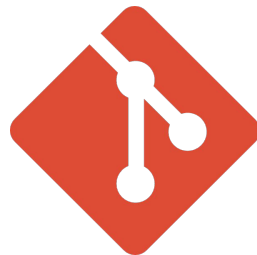
- `git commit -m "the first commit"`

Commit the changes

- `git status`

```
On branch master
nothing to commit, working tree clean
```

Amend to a Commit



- `git commit --amend -m "your new msg"`

Maybe you forgot to add a file

The above command will amend
the added change to the last commit

Add & Commit



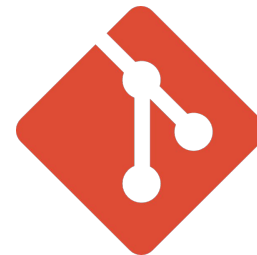
- `git commit -a -m "the first commit"`

Add the changes & commit the changes in one line

But , notice that this command doesn't add new files

It only works with the changes that made inside the files itself

Git Logs

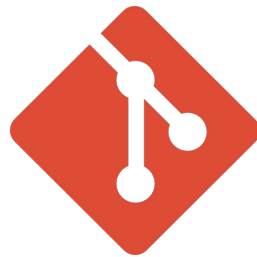


- `git log`

```
commit e0940f8439b5e55a24e09c6bdd1aeacfb8b1cf48 (HEAD -> master)
Author: Ahmedsamymahrous <asamy0037@gmail.com>
Date:   Sun Nov 21 08:59:23 2021 +0200

    the first commit
```

Git Diff



- `git diff`

Show the **unstaged** differences since the last commit

- `git diff --staged`

Show the **staged** differences since the last commit

Unstaging changes



- `git restore --staged script.py`

To unstage the changes

Undoing a Commit



- `git reset --soft HEAD^`

Delete the last commit and git back to the staging area

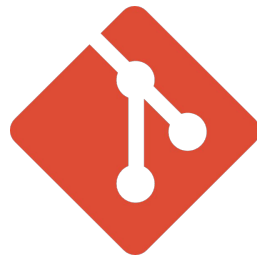
- `git reset --soft HEAD^^`

Delete the last **2** commits and git back to the staging area

- `git reset --hard HEAD^`

Delete the commit and the change from staging area and delete from working copy

Add / Remove Remote Repo



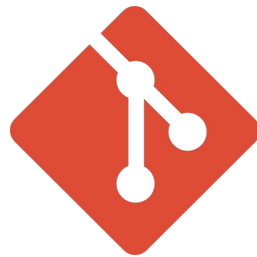
- `git remote add origin`
`https://github.com/any_repo`

To add a remote repository

- `git remote -v`

To list the remote repositories

Push / Pull Remote Repo



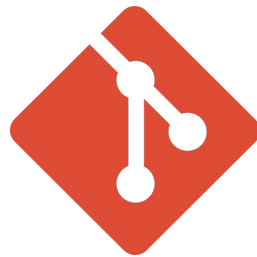
- `git push origin master`

To push the local changes to the remote repo

- `git pull origin master`

To get the changes that made by others

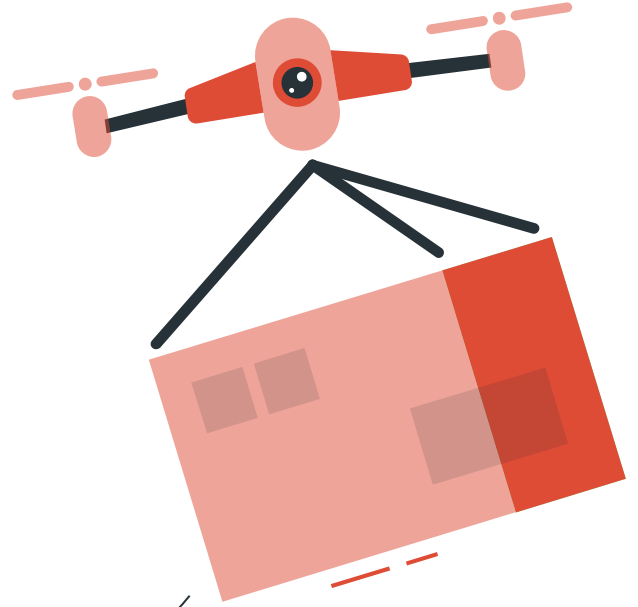
Clone a Remote Repo



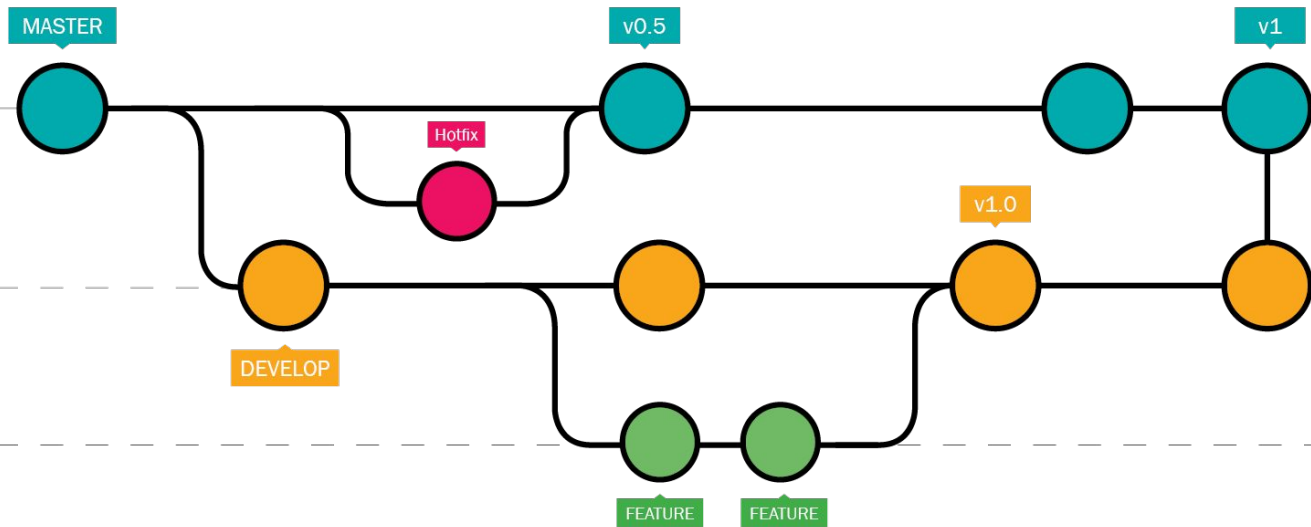
- `Git clone https://github.com/any_repo`

To clone the entire repository to your
local machine in a new directory

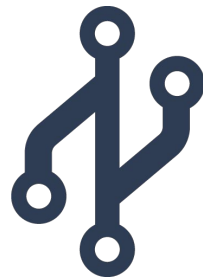
Branching & Rebasing



Branching out



Branching Out



- To make a **new** branch.

```
git branch new_branch_name
```

- To **list** all the branches

```
git branch
```

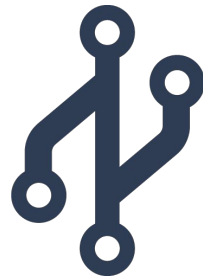
- To **switch** to a branch

```
git checkout branch_name
```

- To create a branch and checkout it in **one step**

```
git checkout -b new_branch_name
```

Create a Remote Branch



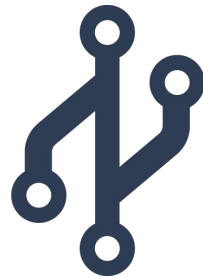
- When you need another people to work on your branch
Then you have to make your branch available remotely

```
git push origin branch_name
```

- To list remote branches

```
git branch -r
```

Remove a Branch



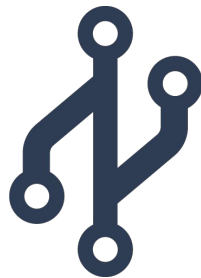
- To delete a **remote** branch

```
git push origin :branch_name
```

- To delete a **local** branch

```
git branch -d branch_name
```

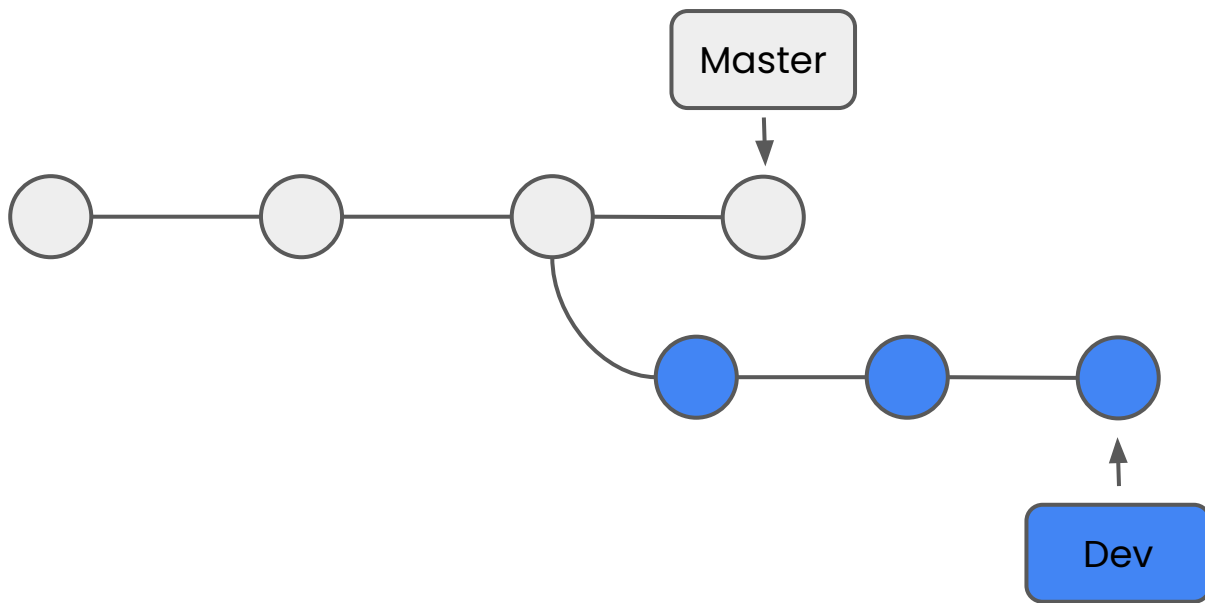
Merging Branches



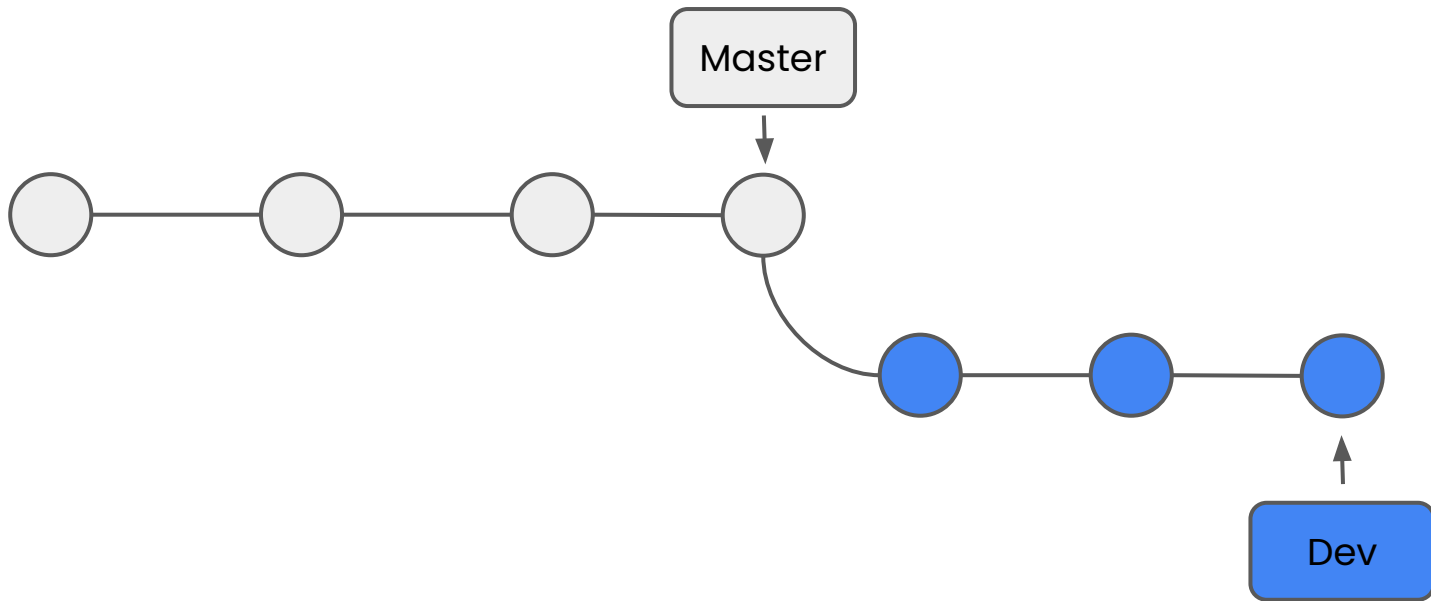
After finishing your work on the branch, you've to **merge** it with the Master branch.

- First, go to the Master branch
`git checkout master`
- Then, merge the two branches with each other
`git merge branch_name`

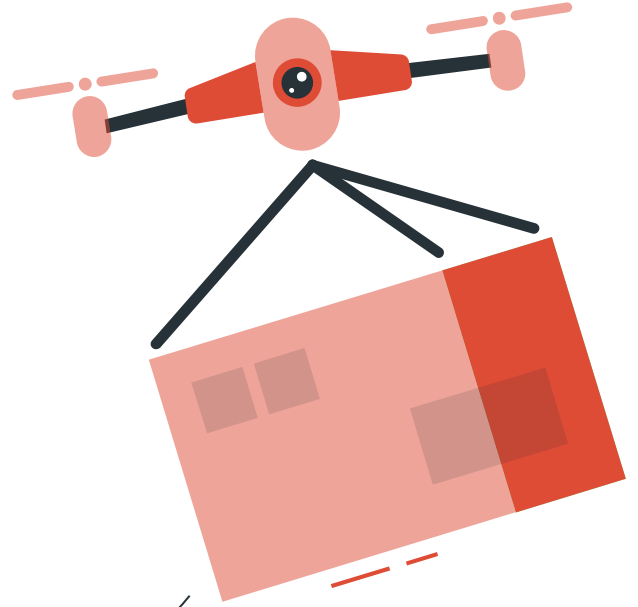
Git Rebase



Git Rebase



Pull Request



Pull Request

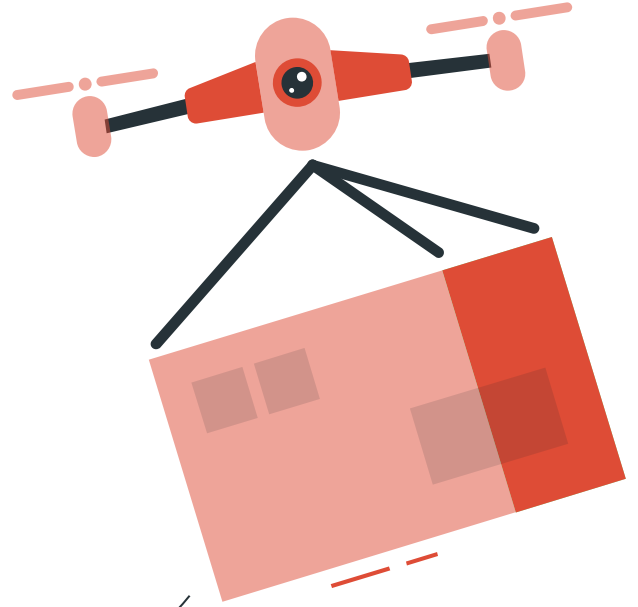
Pull requests let you **tell others** about changes you've pushed to a branch in a repository on GitHub.

Once a pull request is opened, you can discuss and review the **potential changes** with collaborators and add follow-up commits before your changes are merged into the base branch.



**Demo on
GitHub**

Tagging & Versioning



Tagging

- A tag is a reference to a commit – used mostly in release versioning.

Git supports two types of tags:

- Lightweight
- Annotated.

Tags Types

- To create a **lightweight** tag

```
git tag v1.0
```

- To create an **annotated** tag

```
git tag -a v2.0 -m "version 2.0"
```


Push Tags

- To list all tags

```
git tag
```

- To push tags

```
git push origin <tag_name>
```

```
git push --tags
```

Delete Tags

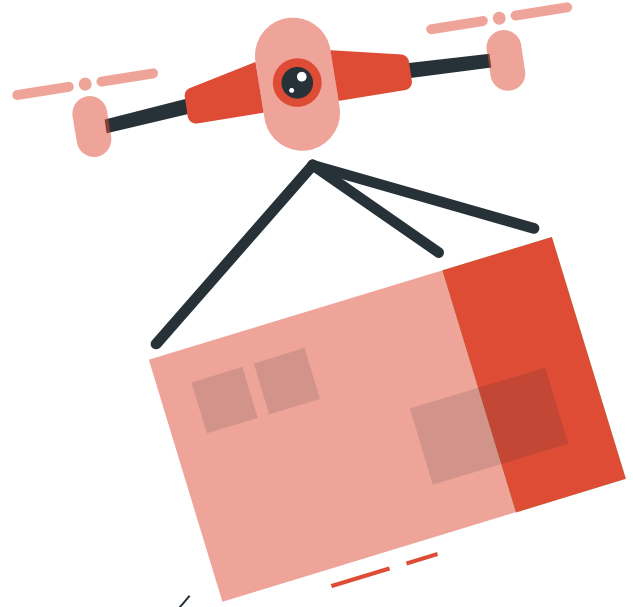
- To delete remote tag

```
git push origin --delete v1.0
```

- To delete local tags

```
git tag -d v1.0
```

Ignoring Files



Ignoring Files

- Often, you will have a class of files that you don't want git to **automatically add** or even show to you as being untracked.
- In such cases you can create a file called **.gitignore** to contains all the unwanted files or directories.

→ cache/

→ logs/*.log

Scenario 1: Creating and Committing

- Participants create a new directory on their computers.
- Inside the directory, they create a text file and add some content.
- They initialize a new Git repository in the directory.
- Participants add the text file to the staging area and commit the changes.
- They modify the content of the file, stage it, and commit again.
- Participants use "git log" to view their commit history.

Scenario 2: Basic Branching and Merging

- Participants clone a sample repository from a remote source.
- They create a new branch called "feature/add-about-page."
- Inside this branch, they modify an existing HTML file to add an "About" page.
- Participants commit their changes on the feature branch.
- They switch back to the main branch and create a new branch called "bugfix/fix-typo."
- Participants correct some typos in the existing content and commit.
- They merge the "bugfix/fix-typo" branch into the main branch.
- Finally, participants switch to the "feature/add-about-page" branch and merge the main branch into it to integrate any changes made.

Scenario 3: Resolving Conflicts

- Participants are provided with a repository and are asked to clone it.
- They are given a file to modify and told to commit the change.
- Meanwhile, an instructor makes changes to the same file on the remote repository.
- Participants try to pull the changes from the remote repository and encounter a conflict.
- They open the conflicting file, resolve the conflict manually, and commit the resolved version.
- Participants push their changes to the remote repository.

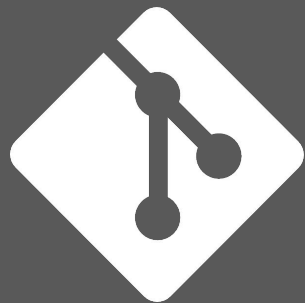
Scenario 4: Collaborating with Remote Repositories

- Participants clone a repository from a remote source.
- They create a new branch named after their username.
- Inside this branch, they modify a text file by adding their name.
- Participants commit the changes and push the branch to the remote repository.
- They open a pull request to merge their branch into the main branch.
- Instructors simulate a review process, providing feedback and requesting changes.
- Participants update their pull request based on the feedback and push the changes.
- The pull request is eventually merged into the main branch.

Scenario 5: Reverting Changes

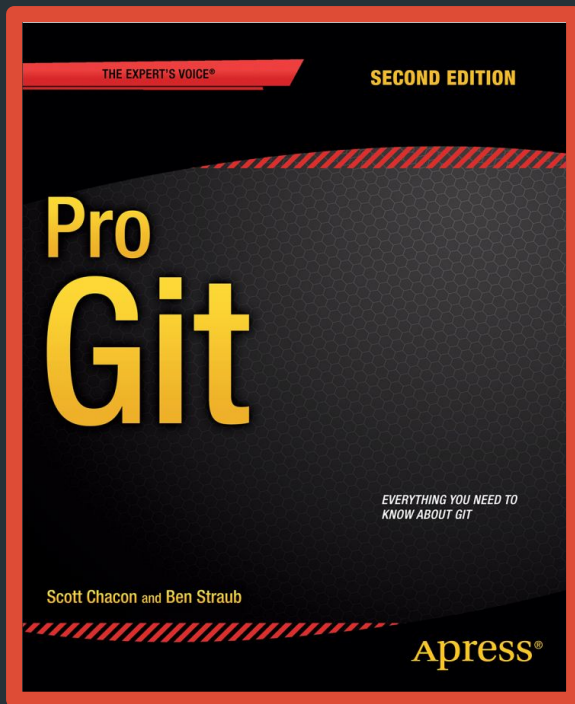
- Participants clone a repository with multiple commits.
- They identify a commit where an unwanted change was introduced.
- Using the commit hash, they use "git revert" to create a new commit that undoes the changes introduced by the identified commit.
- Participants verify that the unwanted change is indeed reverted.
- These beginner-level scenarios provide hands-on experience with essential Git concepts such as creating repositories, making commits, branching, merging, resolving conflicts, collaborating on remote repositories, and reverting changes. They are designed to build a solid foundation in Git for newcomers to version control.

Lab - Bouns

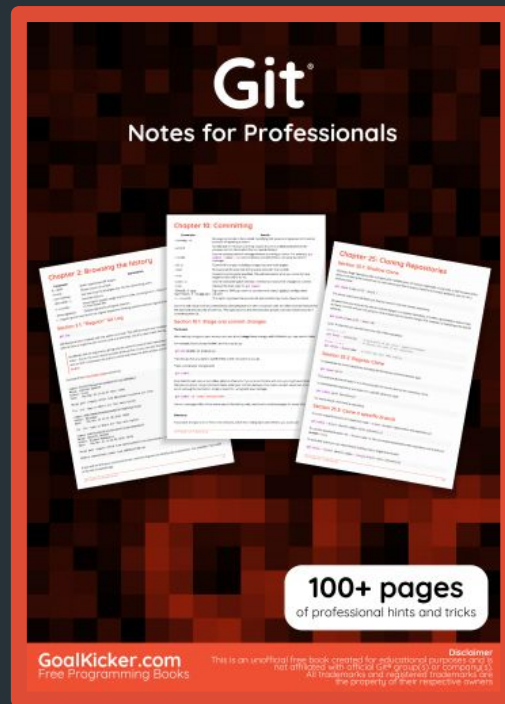


- Create an annotated tag with tagname (v1.7) .
- Push it to the remote repository.
- Tell me how to list tags.
- Tell me how to delete tag locally and remotely.
- Add an image in the README.md file.

RESOURCES



Pro Git – Second Edition



Git – Notes for Professionals

Thanks!

Do you have any questions?