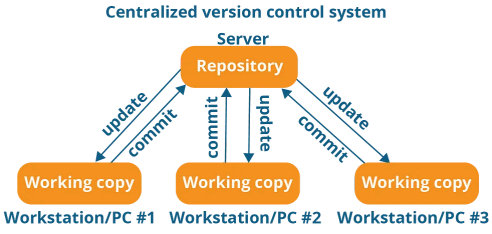
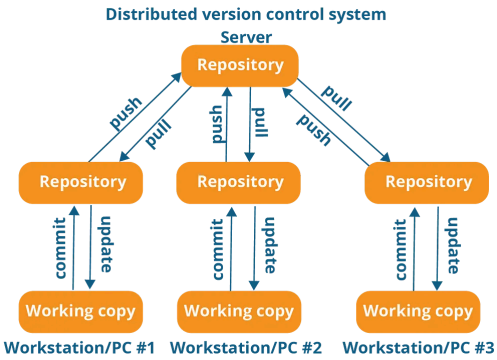
Introduction to Git & GitHub

Git, a Distributed Version Control System

- Created in 2005 for the development of the Linux kernel.
- **Version Control**: Tracks changes made to files over time, letting you revert to previous versions, compare changes or recover lost work.
- **Distributed**: Every developer has a full copy of the repository, including its full history, so they can work offline and sync later or recover from central server failure.
- **Branching and Merging**: Easy to create branches for experimenting or developing features, then merge them back into the main codebase. Allows collaborative work on the same project simultaneously without overwriting each other's work
- **Fast and Lightweight**: Optimized for speed and efficiency, handling large projects with minimal overhead.

Centralized vs Distributed Version Control System





Git - The three stages

Files in a repository go through three stages (phases):

1. **Untracked/Modified**: It is either a new file that Git hasn't seen yet (untracked) or a known file that has changed but not yet been indexed (modified).

   > → Area: *Working Directory or Workspace*

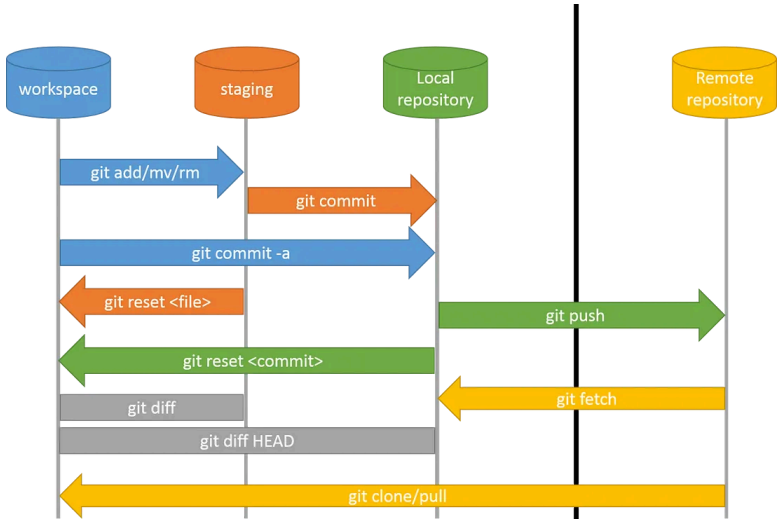
2. **Staged**: The file has been added to git's version control but changes have not been committed.

   > → Area: *Staging or Index*

3. **Committed**: the change has been committed (a new snapshot is created).

   > → Area: Local/Remote repository

Git commands provide the means to move them from one stage to another stage:



Git - Initial Setup

Git needs to be able to identify you and provide a means of contact. When you first start working with Git, you need to do a couple of initial setup steps to configure this information:

```
git config --global user.name "Your name"
git config --global user.email "your.email@yourmailserver.com"
```

In [1]:
```
%%bash
git config --global user.name "Mikel Penagarikano"
git config --global user.email "mpenagar@yahoo.es"
```

The command `git config --list` displays the existing Git config settings:

In [2]:
```
!cd && git config --list
```

```
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
user.email=mpenagar@yahoo.es
user.name=Mikel Penagarikano
```

Non-collaborative Git workflow cycle

1. **Initialize or Clone project**:
   - New Project: Inside the project folder, `git init` creates a new repository
   - Existing Project: Anywhere, `git clone <repository_url>` downloads a copy of a remote repository.

In [3]:
```
!cd ~/tmp && rm -rf my_project && \
    mkdir my_project && cd my_project && git init
```

```
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/jupyter-mpenagaricano/tmp/my_project/.g
it/
```

2. **Make Changes**:
   - Create, edit, remove files...

In [4]:
```
%%bash
cd ~/tmp/my_project
curl https://www.gutenberg.org/cache/epub/10/pg10.txt -o bible.txt 2> /dev/null
cat bible.txt | \
    tr [:upper:] [:lower:] | tr -c '[:alpha:]\n' ' ' | tr ' ' '\n' | \
    sort | uniq -c | sort -nr > words.txt
wc *
ls -l
```

```
 99968  824538 4455996 bible.txt
 12871   25741  207567 words.txt
112839  850279 4663563 total
total 4556
-rw-r--r-- 1 jupyter-mpenagaricano jupyter-mpenagaricano 4455996 Apr 16 15:00 bib
le.txt
-rw-r--r-- 1 jupyter-mpenagaricano jupyter-mpenagaricano  207567 Apr 16 15:00 wor
ds.txt
```

3. **Stage Changes**:
   - Choose which changes you want to include in the next commit and add them to the Staging Area

    - `git add <filename>` Add a specific file
    - `git add .` Recursively add all files in the directory
  - Review changes
    - `git status` Review changes made to the staging

In [5]:
```
!cd ~/tmp/my_project && git add bible.txt && git status
```

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   bible.txt

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

In [6]:
```
!cd ~/tmp/my_project && git add . && git status
```

```
On branch master

No commits yet

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

4. **Commit Changes**:
   - Save the staged changes as a permanent snapshot in your Local Repository.
   - Add a descriptive message explaining what was changed and why.
   - `git commit -m "Your descriptive commit message"`

In [7]:
```
!cd ~/tmp/my_project && git commit -m "My first commit" && git status
```

```
[master (root-commit) 461ee0c] My first commit
 2 files changed, 112839 insertions(+)
 create mode 100644 bible.txt
 create mode 100644 words.txt
On branch master
nothing to commit, working tree clean
```

Most frequent task: commit all changes to existing tracked files.

- `git commit -a` which is equivalent to:
  - `git add -u && git commit`
  - `-u` option: Stages already tracked files (modified/deleted but not new ones)
- If there are new files → `git add . && git commit`

In [8]:
```
!ls ~/tmp/my_project
```

```
bible.txt  words.txt
```

```
In [9]: !cd ~/tmp/my_project && echo "some text" >> bible.txt && git commit -a -m "git t
```

```
[master 1566b68] git test
 1 file changed, 1 insertion(+)
```

```
In [10]: !cd ~/tmp/my_project && echo "some text" > test.txt && git commit -a -m "git tes
```

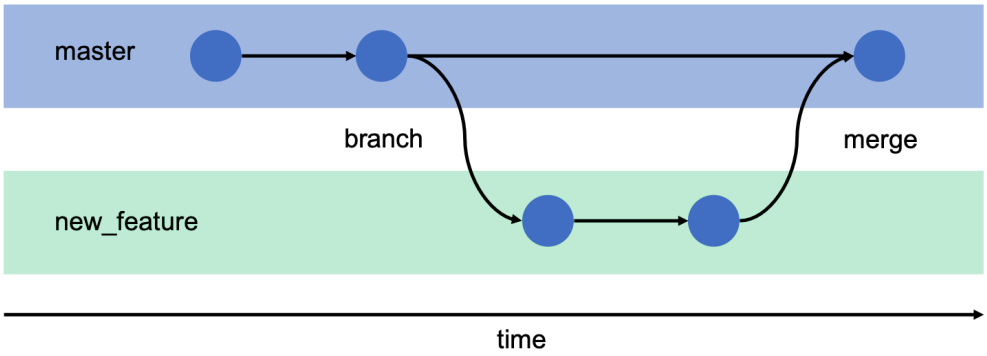
```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test.txt

nothing added to commit but untracked files present (use "git add" to track)
```

```
In [11]: !cd ~/tmp/my_project && git add . && git commit -m "git test 2"
```

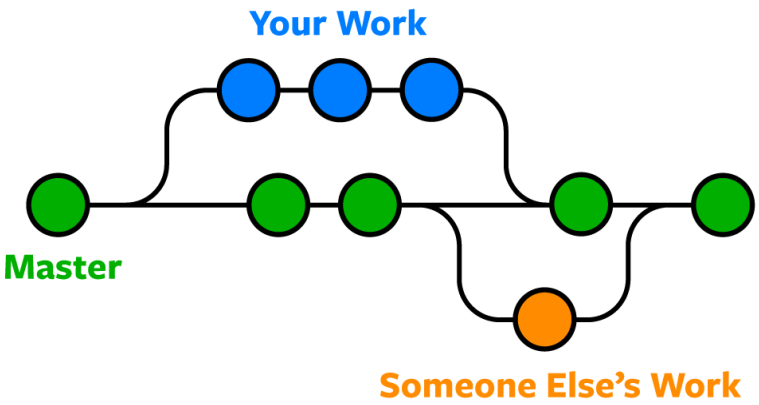
```
[master 68e4179] git test 2
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

Branching & Merging

Git branching and merging are key features that allow to work on different tasks/features without affecting the main codebase.



There can be many branches and all of them (as well as the the master/main branch) can evolve simultaneously.



Branching in Git

- New branches can be created on demand

```
git branch new_branch_name
```

- Workspace can be switched to any existing branch

```
git checkout a_branch_name  # old style
git switch a_branch_name  # new style
```

- Branch creation and switch in a single command:

```
git checkout -b a_branch_name  # old style
git switch -c a_branch_name  # new style
```

In [12]:
```
%%bash
cd ~/tmp/my_project
git switch -c "test1"
echo "some text" > new_feature.txt
git add . && git commit -m "the message"
```

```
Switched to a new branch 'test1'
[test1 75c51b2] the message
 1 file changed, 1 insertion(+)
 create mode 100644 new_feature.txt
```

In [13]:
```
%%bash
cd ~/tmp/my_project
echo "=== ls ===" && ls
git switch master
echo "=== ls ===" && ls
git switch test1
echo "=== ls ===" && ls
```

```
=== ls ===
bible.txt
new_feature.txt
test.txt
words.txt
Switched to branch 'master'
=== ls ===
bible.txt
test.txt
words.txt
Switched to branch 'test1'
=== ls ===
bible.txt
new_feature.txt
test.txt
words.txt
```

Merging in Git

- Combine changes from one branch into another
  - Typically: `feature/bug` branch → `master/main` branch
  - `git merge src_branch_name`: merges the source branch into the current branch

- Steps:
    1. **Find the Common Ancestor**: last common commit between the branches.
    2. **Compare the Changes**
    3. **Apply the Changes**... If possible → **Merge Conflicts**

Merge Conflicts

A **merge conflict** happens whenever there are irreconcilable changes in both branches:

- Different changes to the same lines in a text file
- Different changes to a binary file
- One branch deleted a file that the other branch modified
- Different content in newly added file
- Different changes to file permissions/modes
- Different renaming of a file

→ **merge conflicts must be manually corrected**

In [14]:
```
%%bash
cd ~/tmp/my_project
git switch test1
echo "=== ls ===" && ls
git switch master
echo "=== ls ===" && ls
echo "=== git merge test1 ===" && git merge test1
echo "=== ls ===" && ls
```

```
Already on 'test1'
=== ls ===
bible.txt
new_feature.txt
test.txt
words.txt
Switched to branch 'master'
=== ls ===
bible.txt
test.txt
words.txt
=== git merge test1 ===
Updating 68e4179..75c51b2
Fast-forward
 new_feature.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new_feature.txt
=== ls ===
bible.txt
new_feature.txt
test.txt
words.txt
```

An example of irreconcilable changes (conflicts):

In [15]:
```
%%bash
cd ~/tmp/my_project
```

```
git switch test1
echo "something xxx" >>  new_feature.txt
git add . && git commit -m "the message"
git switch master
echo "something yyy" >>  new_feature.txt
git add . && git commit -m "the message"
```

```
Switched to branch 'test1'
[test1 94c482a] the message
 1 file changed, 1 insertion(+)
Switched to branch 'master'
[master 6b03055] the message
 1 file changed, 1 insertion(+)
```

In [16]:
```
%%bash
cd ~/tmp/my_project
git merge test1 || echo "=== ERROR!!! ==="
```

```
Auto-merging new_feature.txt
CONFLICT (content): Merge conflict in new_feature.txt
Automatic merge failed; fix conflicts and then commit the result.
=== ERROR!!! ===
```

If the merging fails:

1. The merge process is paused
2. The repository is left in a special *Merging* state
3. Non-conflicting files are updated (staged or added to the index)
4. Conflict marks (text) are added to conflicting text files

Two alternatives:

- Manually resolve the conflicts and commit
  - Edit file → `git add <file>` → `git commit`
- Abort the merge
  - `git merge --abort`

Merging state example:

In [17]:
```
%%bash
cd ~/tmp/my_project
cat new_feature.txt
git switch test1 && cat new_feature.txt || echo "=== ERROR!!! ==="
```

```
some text
<<<<<<< HEAD
something yyy
=======
something xxx
>>>>>>> test1
fatal: cannot switch branch while merging
Consider "git merge --quit" or "git worktree add".
=== ERROR!!! ===
```

Manually resolve the conflicts and commit

In [18]:
```
%%bash
cd ~/tmp/my_project
head -1 new_feature.txt > tmp_file
echo "something zzz" >> tmp_file
mv tmp_file new_feature.txt
cat new_feature.txt
```

```
some text
something zzz
```

In [19]:
```
%%bash
cd ~/tmp/my_project
git add new_feature.txt
git commit -m "the message"
```

```
[master fabbe65] the message
```

In [20]:
```
%%bash
cd ~/tmp/my_project
git switch test1 && cat new_feature.txt
git switch master && cat new_feature.txt
```

```
Switched to branch 'test1'
some text
something xxx
Switched to branch 'master'
some text
something zzz
```

Best practices instead of direct merging

First, let's create a conflict:

In [21]:
```
%%bash
cd ~/tmp/my_project
git switch test1
echo "something xxx" >> new_feature.txt
git add . && git commit -m "the message"
git switch master
echo "something yyy" >> new_feature.txt
git add . && git commit -m "the message"
```

```
Switched to branch 'test1'
[test1 29e94cf] the message
 1 file changed, 1 insertion(+)
Switched to branch 'master'
[master 7af0655] the message
 1 file changed, 1 insertion(+)
```

Best practices 1: check the merge without committing

In [22]:
```
%%bash
cd ~/tmp/my_project
git merge --no-commit --no-ff test1
git merge --abort
```

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

- `git merge --no-commit --no-ff` → does not commit or fast forward
- `git merge --abort` → abort the merge

Best practices 2: merge in a temporary branch

In [23]:
```
%%bash
cd ~/tmp/my_project
git switch master
git switch -c tmp_branch_check
git merge test1 || git merge --abort
git switch master
git branch -D tmp_branch_check
```

```
Already on 'master'
Switched to a new branch 'tmp_branch_check'
Auto-merging new_feature.txt
CONFLICT (content): Merge conflict in new_feature.txt
Automatic merge failed; fix conflicts and then commit the result.
Switched to branch 'master'
Deleted branch tmp_branch_check (was 7af0655).
```

- `git merge test1 || git merge --abort` → merge and abort if it fails
- `git branch -D tmp_branch_check` → force-delete the branch

Working with Remote Repositories

Remote repositories serve as common points for developers to synchronize their work.
Key concepts:

- **Remote** → URL references/aliases
- **Cloning** → downloading a remote repository for the first time
- **Fetching** → downloading new data from a remote repository
- **Pulling** → downloading and merging new data from a remote repository
- **Pushing** → uploading and merging new data to a remote repository

Remote

- A reference (a name associated with a URL) that points to another copy of the repository (hosted in GitHub, GitLab, private server...).
- A cloned repository already has a remote called `origin`
- `git remote -v` → display configured remotes

In [24]:
```
!cd ~/stats_for_ai && git remote -v
```

```
origin  https://github.com/iurteagalab/stats_for_ai (fetch)
origin  https://github.com/iurteagalab/stats_for_ai (push)
```

Cloning

- Usually the first step.
- Downloads the entire repository from the remote URL.
- Creates a new directory, initializes a full local Git repository, sets up the `origin` remote and checks out the default branch (`main` or `master`).
- Creates *remote-tracking branches* (`origin/main` or `origin/master` and all other branches)
- `git clone <repository-url>`

In [25]:
```
%%bash
cd ~/tmp
rm -rf Programming-for-AI
git clone https://github.com/mpenagar/Programming-for-AI
cd Programming-for-AI
git remote -v
```

```
Cloning into 'Programming-for-AI'...
origin  https://github.com/mpenagar/Programming-for-AI (fetch)
origin  https://github.com/mpenagar/Programming-for-AI (push)
```

Fetching

- Downloads from a remote (`origin` by default) all the new data.
- Updates *remote-tracking branches* (`origin/main` or `origin/master` and all other branches)
- Does not change any local branch.
- `git fetch origin` → update remote-tracking branch from `origin`

An updated *remote-tracking branch* can be merged like any other branch

- `git merge origin/main` → merge the remote changes

Pulling

- Fetches a remote and then merges a branch (`git fetch` + `git merge`)
- There is no way to check conflicts
- `git remote -v` → display configured remotes

Pushing

- Updates a branch of the remote repository.
- Remote branch cannot be ahead of local branch
  - Local branch must include all the commits present on the remote branch.
  - `git pull` (or better, `git fetch` + check + `git merge`) before `git push`
- Requires write permissions on the remote repository
- `git push <remote> <branch>` → use local branch to update remote branch
- `git push --force <remote> <branch>` → **(DANGER!)** force push (if remote is ahead)

GitHub

- A web-based platform for hosting Git repositories.
- Built on top of Git, but adding many features
- Git + social + cloud + collaboration tools

Feature	Git	GitHub
What it is	Version control system	Platform for hosting Git repositories
Where it runs	Locally (your computer)	In the cloud (GitHub servers)
Creator	Linus Torvalds	GitHub Inc. (now owned by Microsoft)
Interface	Command-line tool	Web interface + API
Works without internet	✅ Yes	❌ No
Repo hosting	❌ No	✅ Yes
Collaboration tools	❌ No	✅ Yes (PRs, issues, reviews, etc.)
Access control	❌ N/A	✅ Teams, organizations, private repos
CI/CD & automation	❌ No	✅ GitHub Actions
Social features	❌ None	✅ Stars, followers, forks
Alternatives	N/A	GitLab, Bitbucket, Azure Repos, etc.