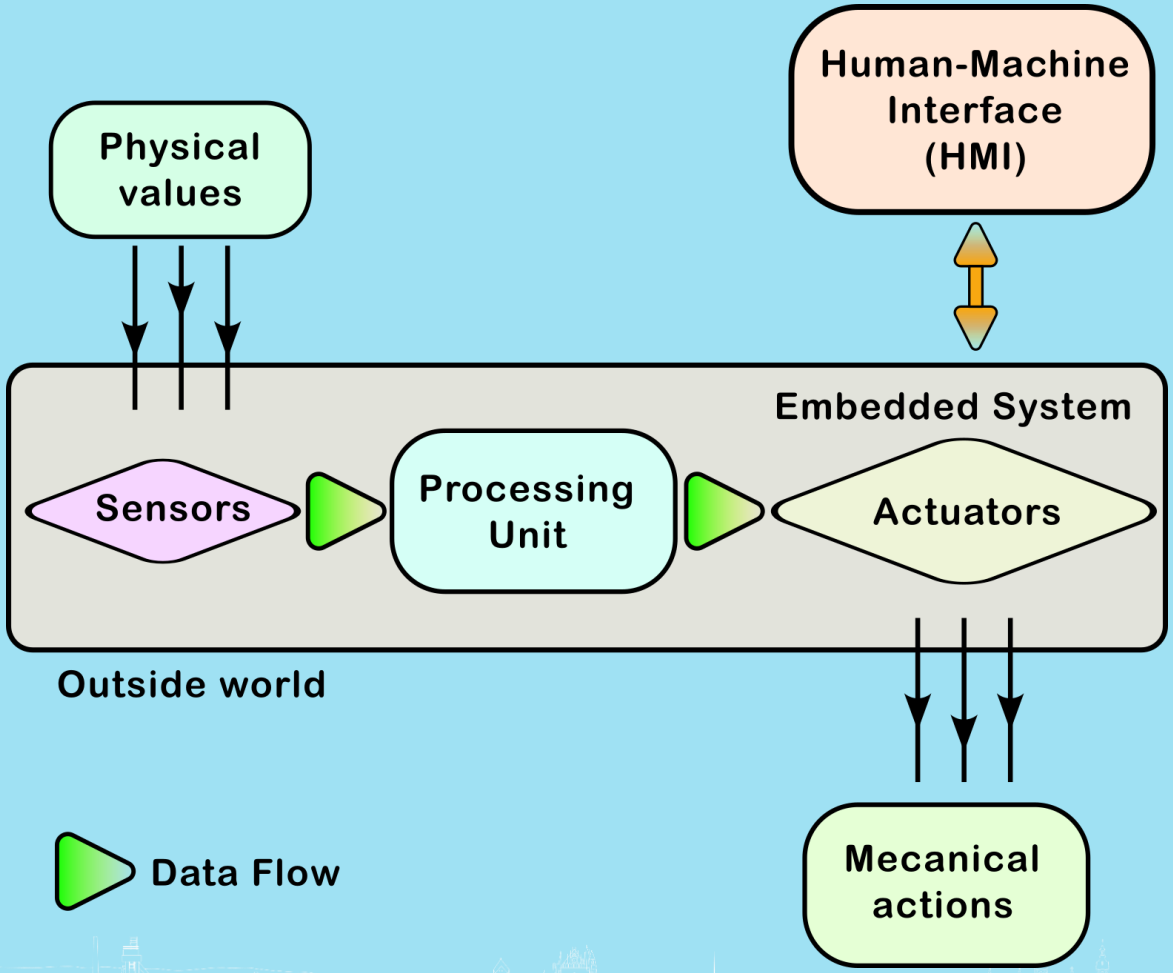
Collaborative work tools

Yann MORERE
LCOMS – SciFA - Université de Lorraine

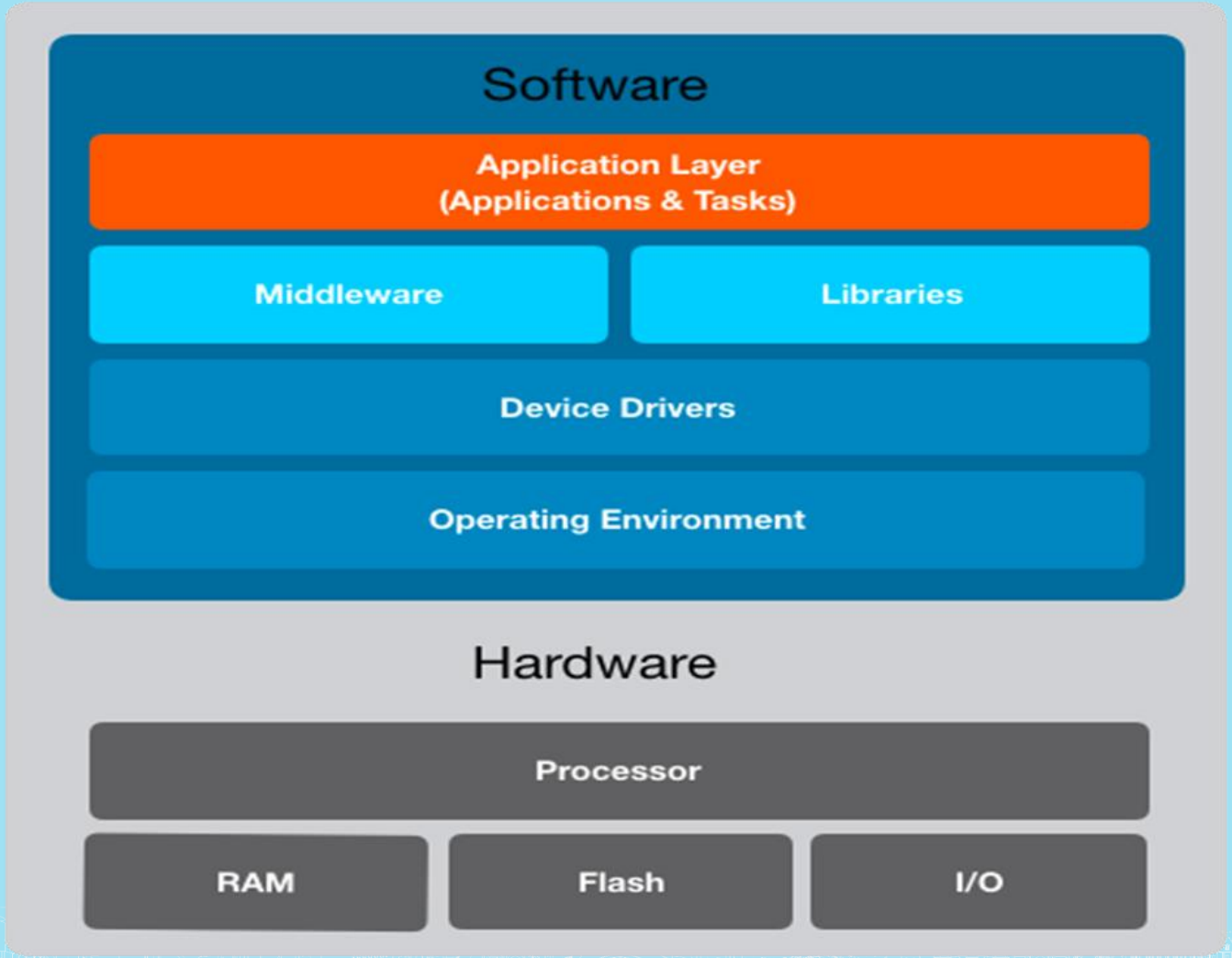
https://zenodo.org/records/21780603

- Understand:
  - Project versioning
  - The basic concepts of git
  - Collaborative work
  - Project management using gitlab
- During the practical work, learn to use:
  - The basic git commands (workstation side)
  - The main features of gitlab (server side)
  - Good practices

- Why a collaborative tool?
- History of VCS
- The concepts of git
- Gitlab features



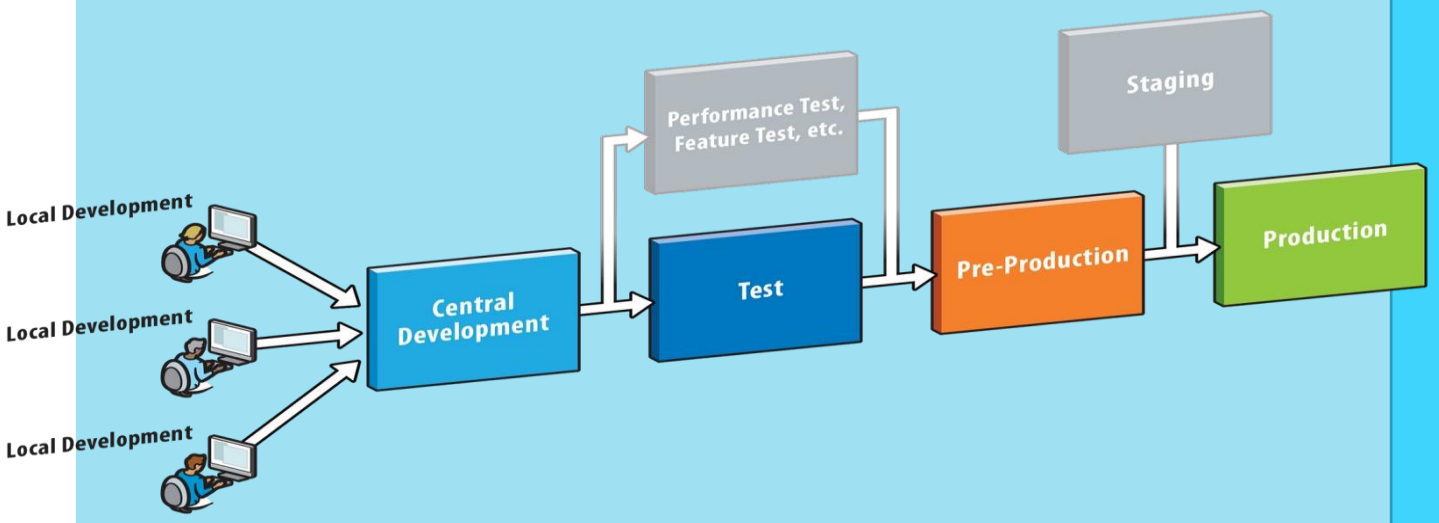
- Definition: An embedded computer system receives information from the outside world by means of **sensors** (or sensors). **Its programmable board stores and processes this information by the processing unit** (the microprocessor) and then sends information back to the outside world via **actuators** (or actuators).



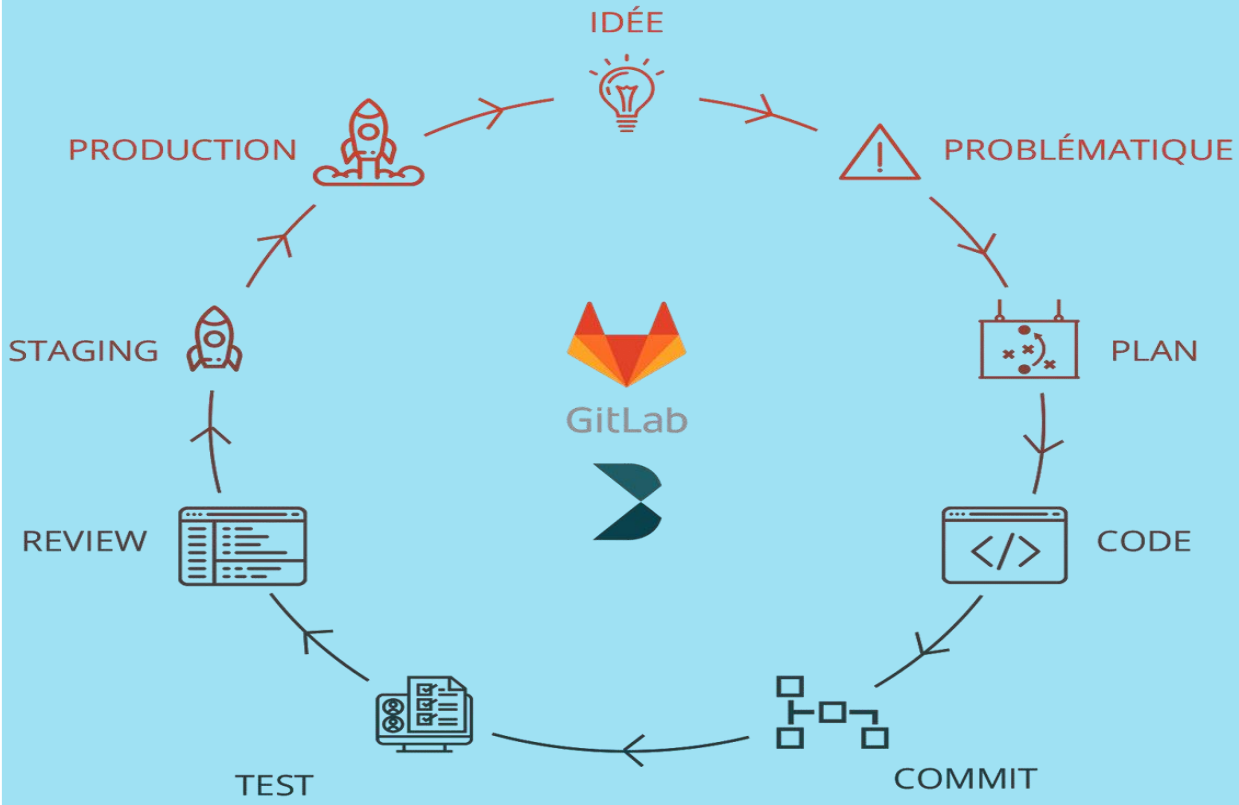
Embedded software consists of several layers, with the upper layers implementing increasingly abstract features.

The **life cycle of a software component** can be divided into 4 phases:

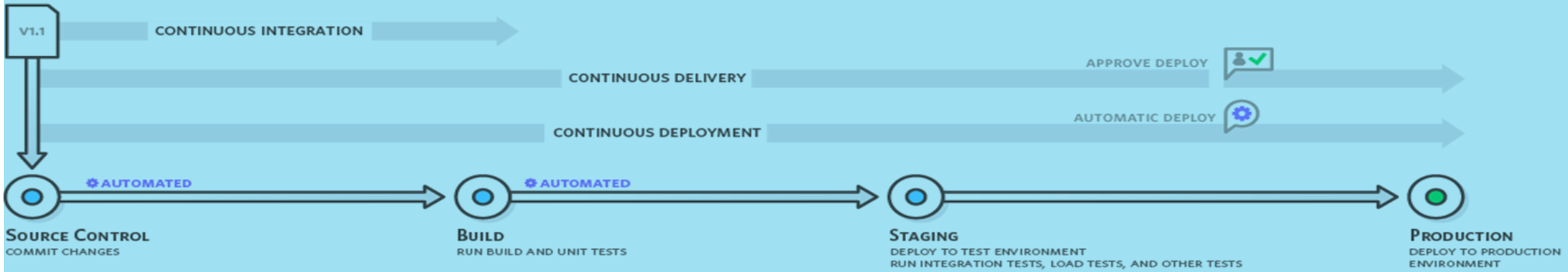
- The creation of the tool (design, production and deployment)
- Its use by users (administration, use)
- Its update and evolution (maintenance)
- The decision to replace the tool (end of life)



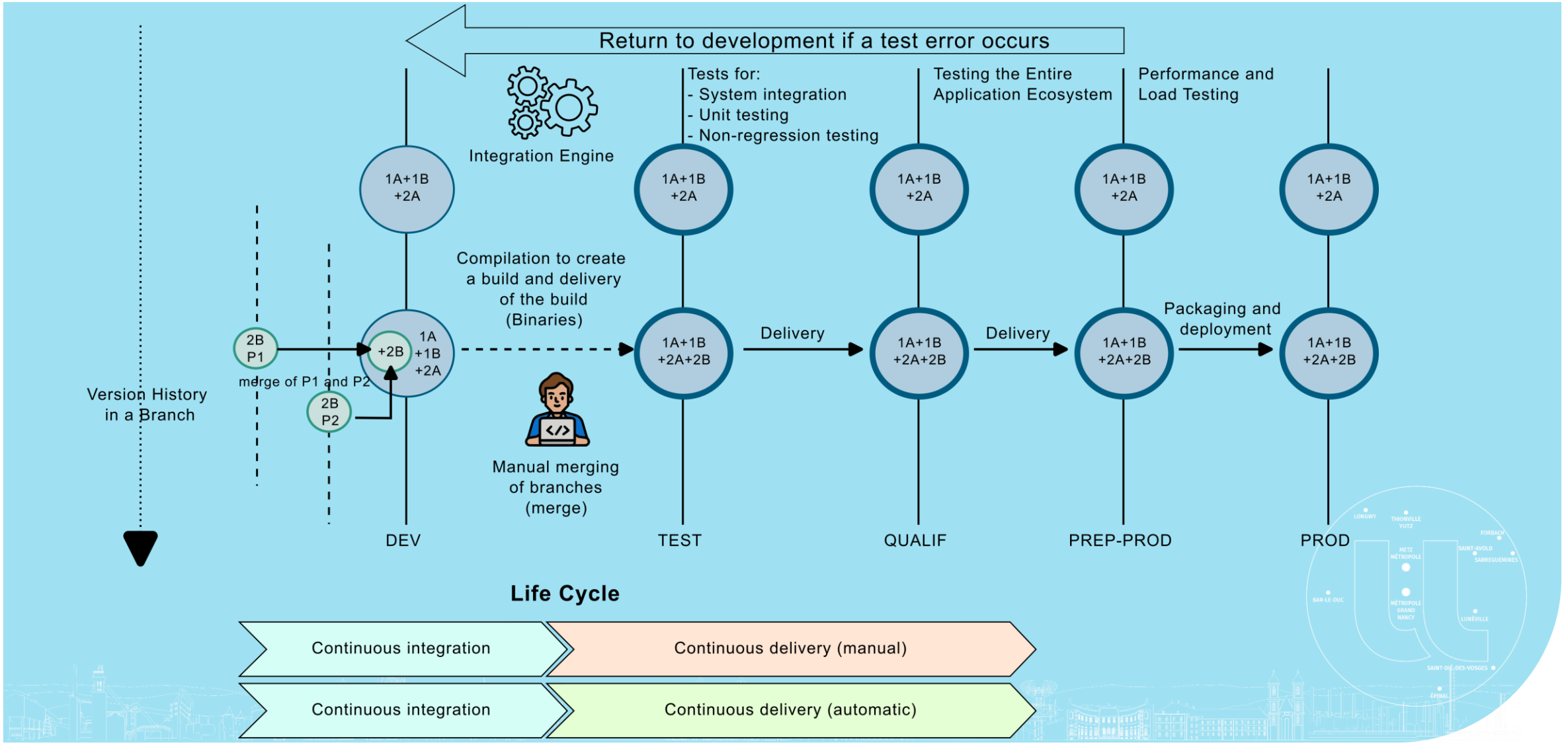
The 6 possible environments:
- **Development** also called developer's computer
- **integrated development** a remote environment used by developers to integrate their code so that they can check if they have broken someone else's code, or if someone else has broken their code
- **Test Environments**: Environments used for manual and/or automated testing
- **staging environment** used for acceptance testing before pushing the new
- **Canary environment**, a special type of environment used to test new features with a small group of users (commonly used to test things internally before reaching a wider audience)
- **Production environment** The environment that our users actually use

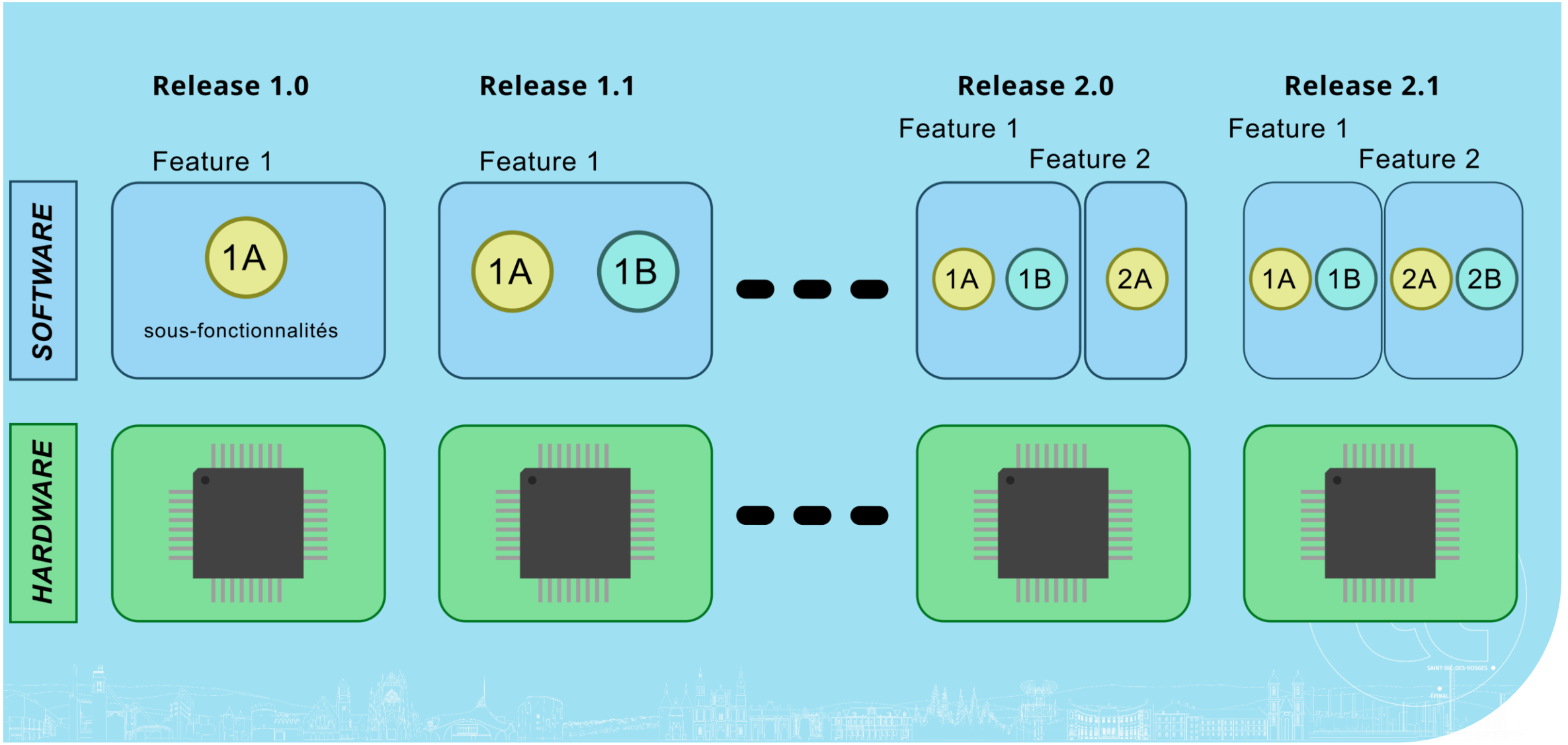


A **collaborative tool** such as GitLab allows everyone to follow the project, from the end customer who knows nothing about code to the developers of the application.
This makes it possible to manage the entire creative process from the customer idea to the delivery of the product.
Once the expression of need has been issued and the development plan has been validated, the code is developed and posted on Gitlab.
Each modification is made in the form of a "commit" and thus gives rise to tests of the code for all users.
Depending on the anomalies detected, it is reviewed "review".
The validated code is then put into pre-production "staging" to finally be deployed.



Continuous integration refers to the unit creation and testing stages of the software release process. Each revision applied triggers an automated creation and testing process.
With continuous delivery, code changes are automatically applied, tested, and prepared for production.
Continuous delivery extends the principle of continuous integration by deploying all code changes to a test environment and/or production environment after the build stage.


Return to development if a test error occurs
Integration Engine
Tests for:
- System integration
- Unit testing
- Non-regression testing
Testing the Entire Application Ecosystem
Performance and Load Testing
1A+1B +2A
Compilation to create a build and delivery of the build (Binaries)
2B P1
2B P2
merge of P1 and P2
+2B
1A +1B +2A
Manual merging of branches (merge)
1A+1B +2A+2B
Delivery
Delivery
Packaging and deployment
Version History in a Branch
DEV
TEST
QUALIF
PREP-PROD
PROD
Life Cycle
Continuous integration
Continuous delivery (manual)
Continuous integration
Continuous delivery (automatic)


Release 1.0
Release 1.1
Release 2.0
Release 2.1
Feature 1
Feature 1
Feature 1
Feature 2
Feature 1
Feature 2
SOFTWARE
1A
sous-fonctionnalités
1A
1B
1A
1B
2A
1A
1B
2A
2B
HARDWARE

Version control system(***version control system***)

- A set of software tools for:
  - Memorize and find different versions of a project.
  - Facilitate collaborative work.
- Originally developed by Linus Torvalds to facilitate Linux kernel development.
  - Free / open source software.
  - Available on all platforms.

Git-based web app.

GitLab

- Allows you to manage:
  - The git project lifecycle.
  - Project participants (roles, groups, etc.).
  - Communication between these participants.

Developed by a private company

- The "community edition" is free.
- A free online service (like github)

Postulate: need a system to manage the Software Project Lifecycle.

- Manage versions.
- Manage developers (access rights).
- Manage issues (bug reports, suggestions)
- evolution, etc.) .
- Create online documentation for a project.
- Automatic build, test, and deploy.

Software Development

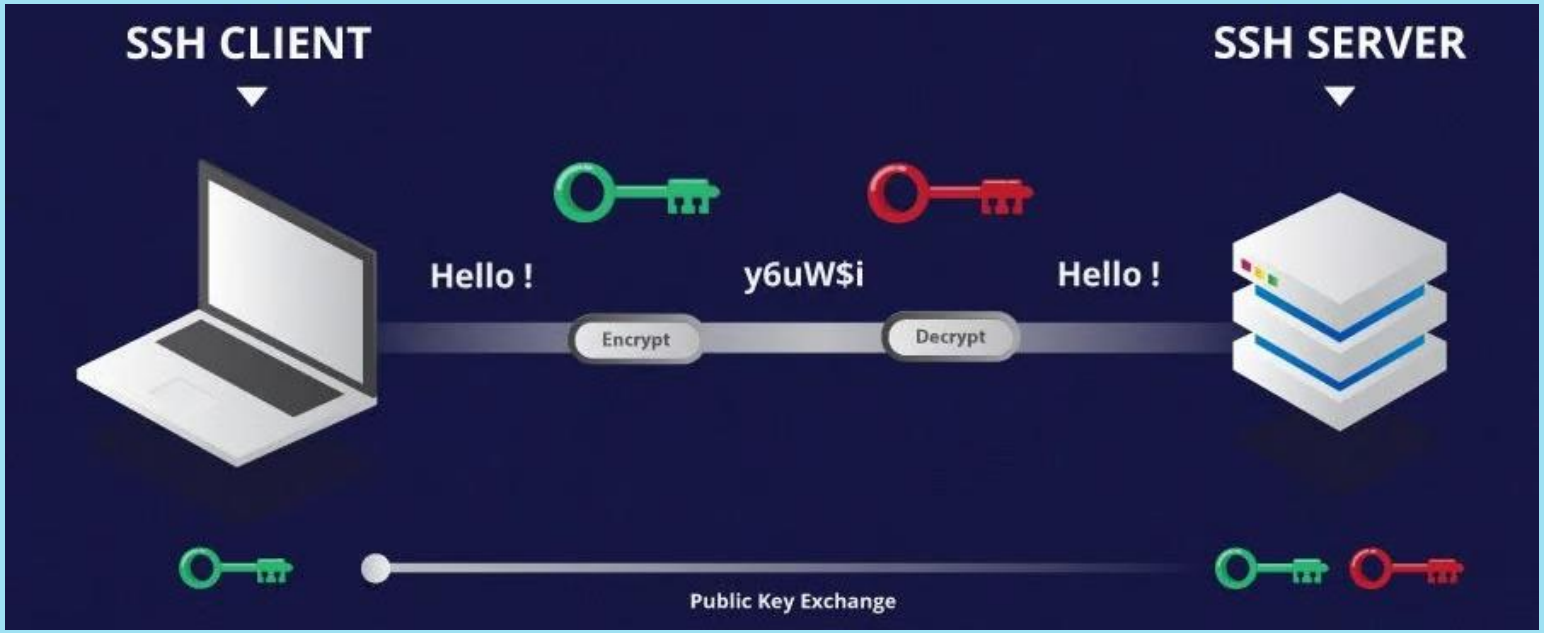
- Designed to manage software development...

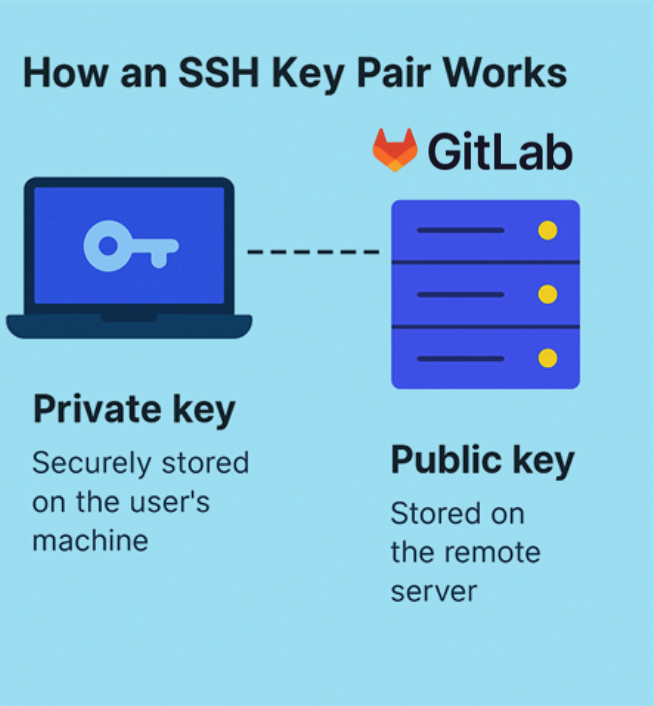
Writing documents

- Articles (e.g. latex sources).
- Web pages (e.g. HTML, markdown, etc.).
- Any type of text document (ascii format).

Archive binaries (use the **git-lfs**)

- executable and "released" libraries
- Files generated by specific editors (e.g. CAD, Word).





Secure Shell (SSH) is both a computer program and a secure communication protocol. The connection protocol requires an exchange of encryption keys at the beginning of the connection. Thereafter, all TCP segments are authenticated and encrypted.

You need to generate an SSH key pair via the: **ssh-keygen**

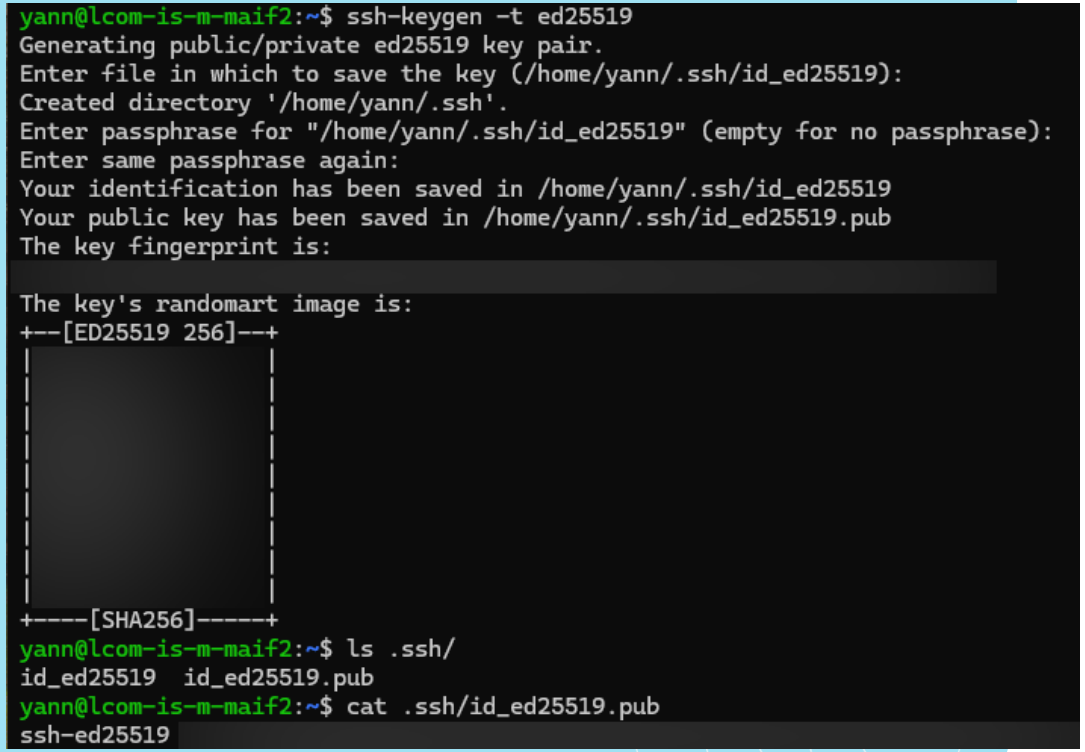
At this point, two keys will be generated in your default 'home' folder:

- Linux: **/home/[user]/.ssh**
- **Warning: The ssh folder is a hidden folder**

The following files can then be found in the ***.ssh*** folder:

- **id_rsa**: **private key to keep on your PC and not to share**
- **id_rsa.pub**: **public key to send to the machines you communicate with**

1. Create an SSH key pair with **ssh-keygen -t ed25519**
2. Make a key backup
3. Use **git clone** in SSH to retrieve the project
4. Declare the **public key on your gitlab account**



Exercise
Git

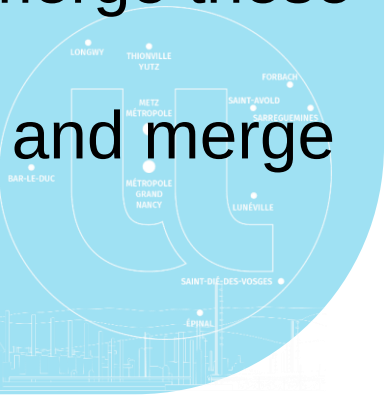
GitLab
Collaborate • Review • Deploy

1. Create account on https://www.gitlab.com, log in
2. Create a first **blank project**
3. Create a helloworld.txt file localy (hard drive)
4. Upload helloworld.txt in the created project
5. Verify that the git client is installed on your machine
6. Create an EX01 directory in your home
7. Use git clone in ssh to recover the project

```
yann@lcom-is-m-maif2: ~/Gi
yann@lcom-is-m-maif2:~$ nano HelloWorld.txt
yann@lcom-is-m-maif2:~$ pwd
/home/yann
yann@lcom-is-m-maif2:~$ mkdir GitlabPractical
yann@lcom-is-m-maif2:~$ cd GitlabPractical/
yann@lcom-is-m-maif2:~/GitlabPractical$ git clone git@gitlab.com:ian57/myfirstgitlabproject.git
Cloning into 'myfirstgitlabproject'...
The authenticity of host 'gitlab.com (172.65.251.78)' can't be established.
ED25519 key fingerprint is: SHA256:eUXGGm1YGsMAS7vkcx6JOJdOGHPem5gQp4taiCfCLB8
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'gitlab.com' (ED25519) to the list of known hosts.
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (6/6), done.
yann@lcom-is-m-maif2:~/GitlabPractical$ ls
myfirstgitlabproject
yann@lcom-is-m-maif2:~/GitlabPractical$ cd myfirstgitlabproject/
yann@lcom-is-m-maif2:~/GitlabPractical/myfirstgitlabproject$ ls
HelloWorld.txt  README.md
yann@lcom-is-m-maif2:~/GitlabPractical/myfirstgitlabproject$
```

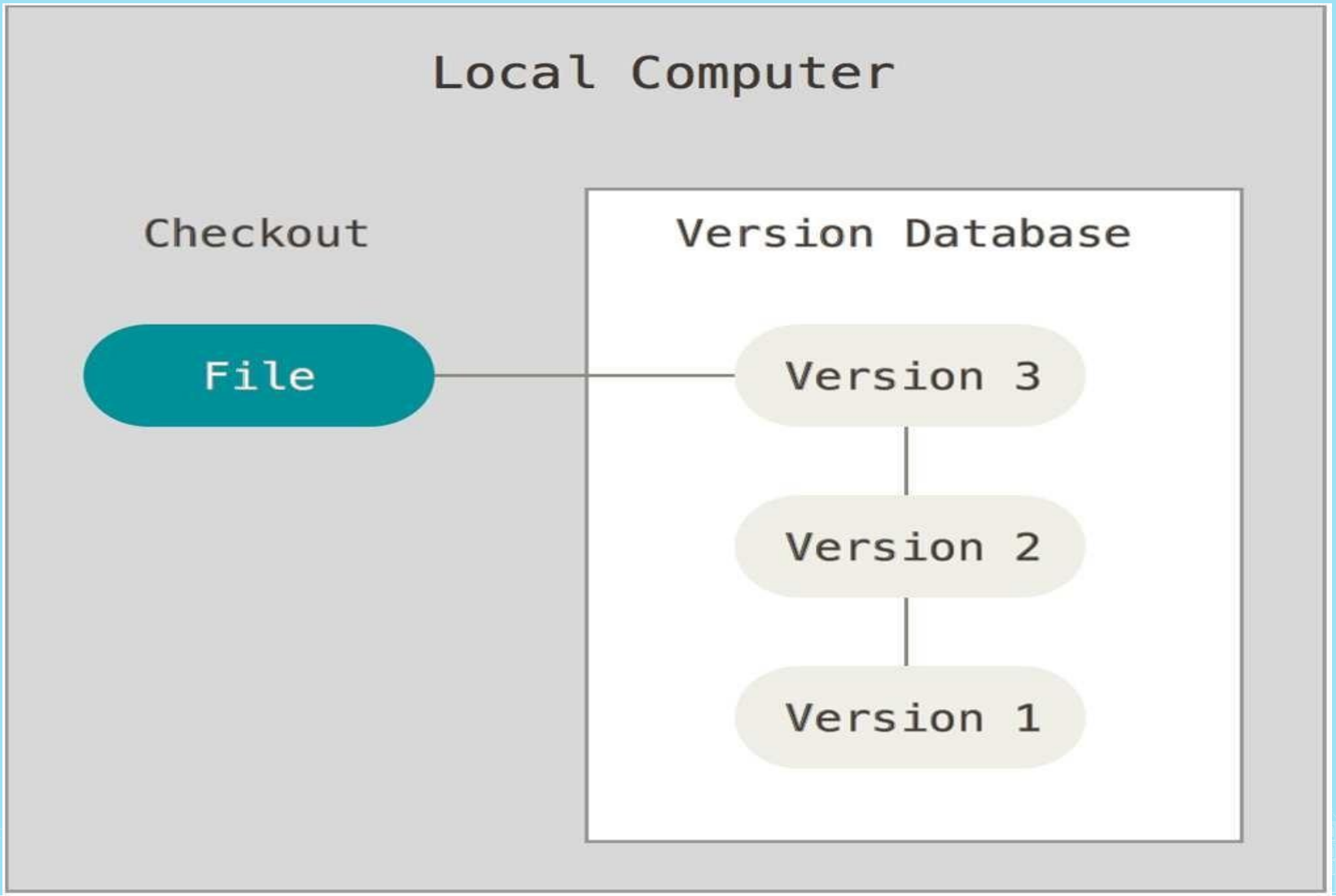
Simple definitions:

- **Release** = content of the project at some point in its lifecycle.
- **Repository** = the history of the project, containing all its versions.
- **Branch** = variant of a project.
- During the development of a project, it is necessary to:
  - Save the **versions** of the project in a **repository**, find a specific version (e.g. latest version without BUG).
  - To work on different variants (**branches**) in parallel, to merge these variants.
  - Share changes made between developers (understand and merge changes made by others).



The ancestor of the VCS: the CPOLD methodology

- **repository** = root folder of the project.
- Each folder/file is associated with a name + an extension.
  - Name = name of the item contained in the project
  - Extension = the version of the file/folder.
- Based on the **cp** command:
  - cp file1.old //"old" version saved
  - cp -R folder folder.1.4 //assumes version 1.3 exists!
- Sharing changes:
  - via a classic file server (or any medium)
  - Merging changes is done manually.


Local Computer
Checkout
File
Version Database
Version 3
Version 2
Version 1

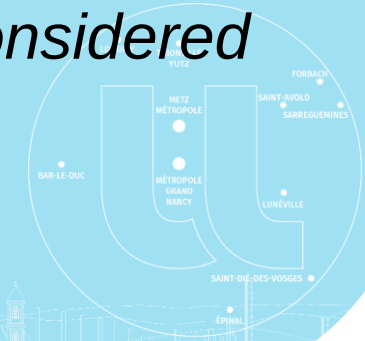
CPOLD : limitations

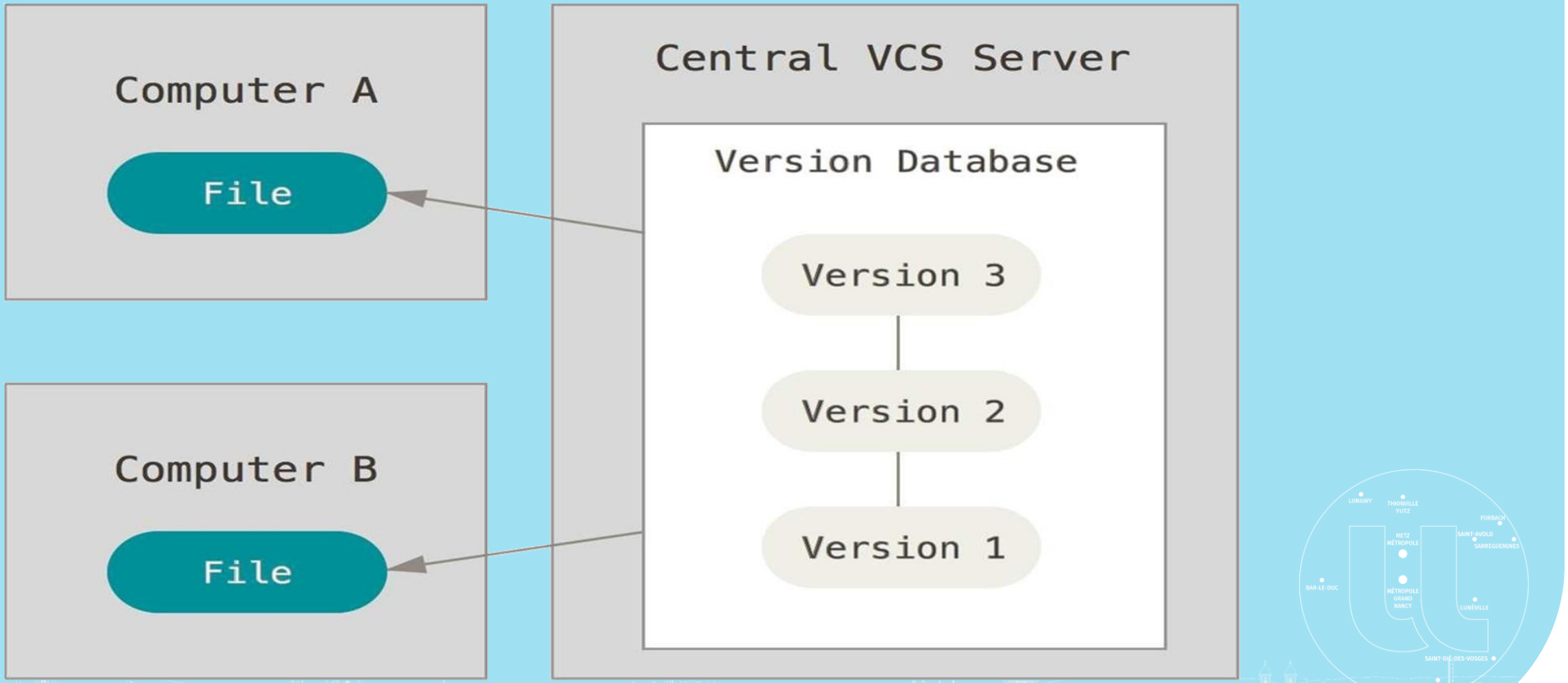
- Naming convention for extensions must be strictly followed.
- Suboptimal in terms of disk space
  - each version = full copy existing code + additions.
- Sharing changes and merging variants is **very laborious**.

Need tools to automate, simplify, and make version control robust.

Centralized VCS: CVS, subversion, etc.

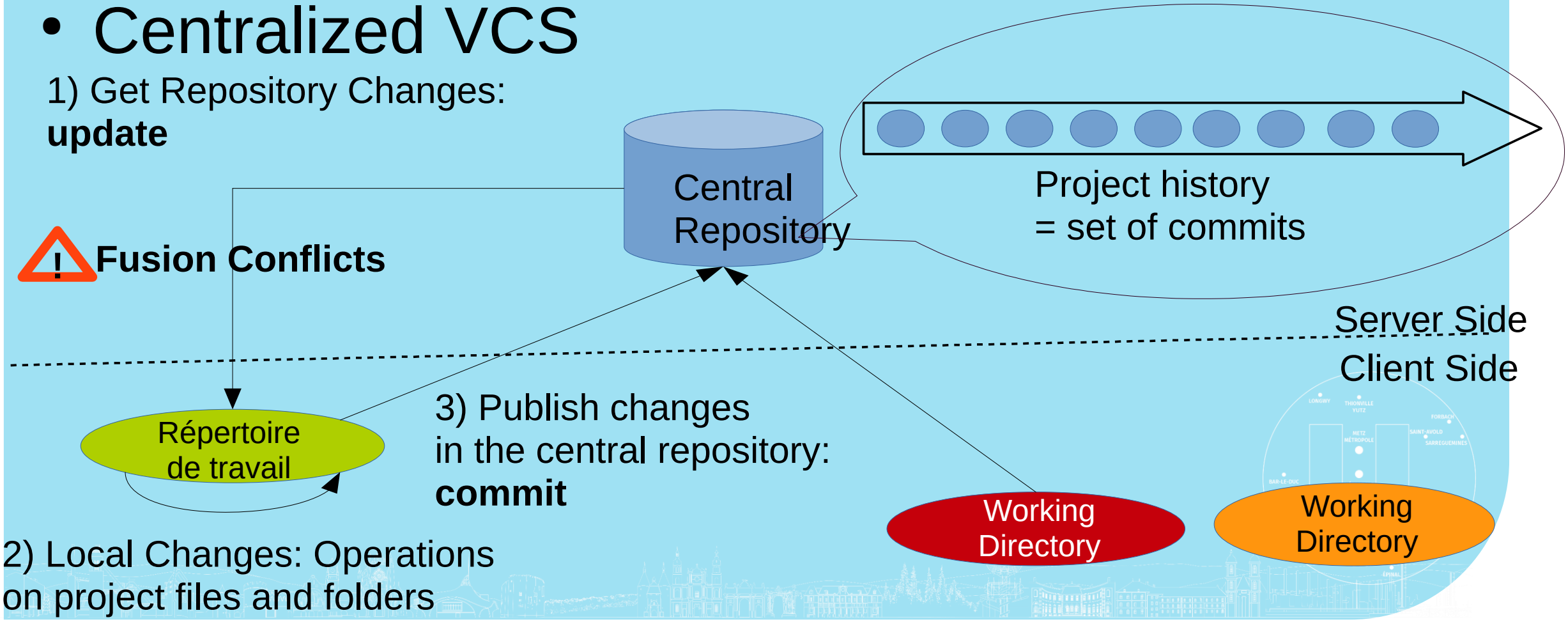
- Software tools based on a client/server approach.
  - Server = a file server that **contains repositories** and manages user access.
  - Client = a workstation that contains a **working folder** connected to the repository.
- Principle: only **changes** are saved.
  - ***Commit****: A set of changes to files/folders in the project, considered atomic (indivisible).*
  - ***repository*** *= commit history*.




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

- Centralized VCS

1) Get Repository Changes:
update



Centralized VCS: advantages

- Optimal size on disk
  - Version = changes made (commit) since previous version.
- No need for conventions.
- Version sharing is automated.

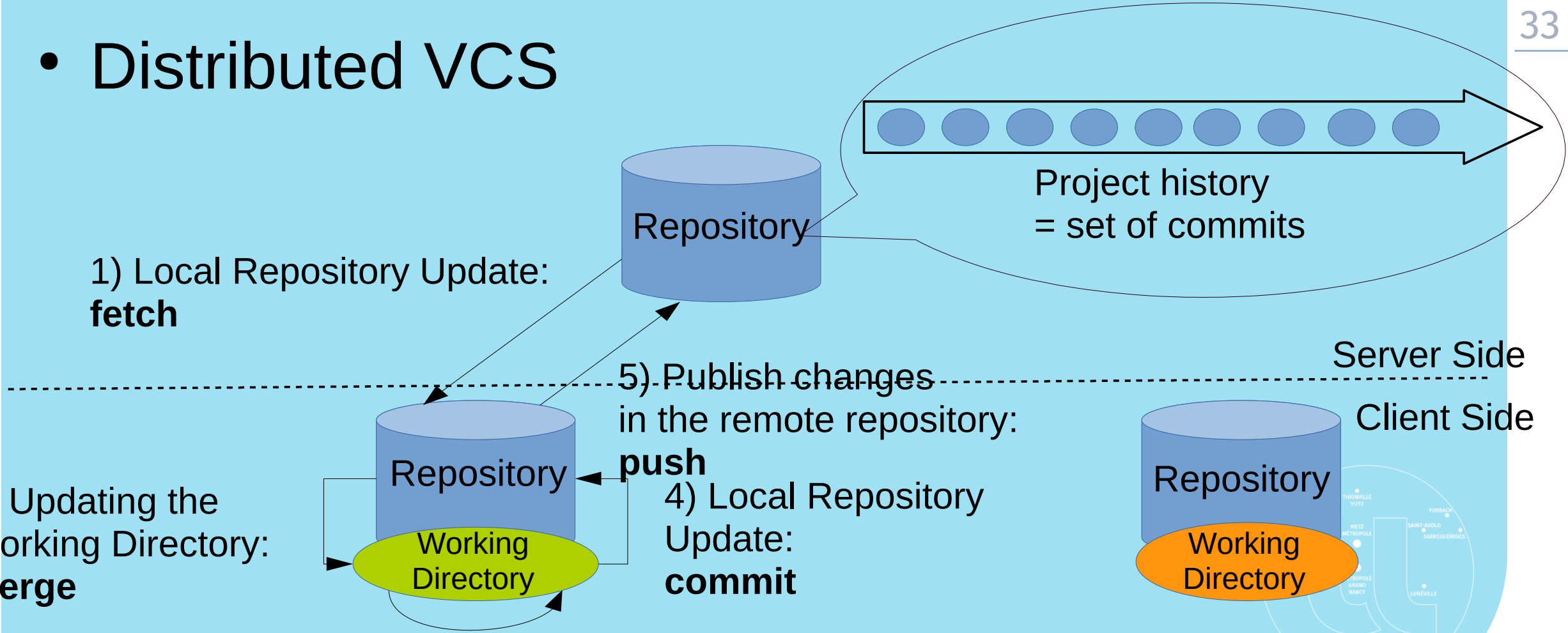
Centralized VCS: Limitations

- Difficult offline work: To create commits/merge changes, you need to synchronize with the server.
- Lack of robustness: Losing the central repository (e.g. data corruption) = losing all.

Distributed VCS: git, Bazaar, mercurial, etc.

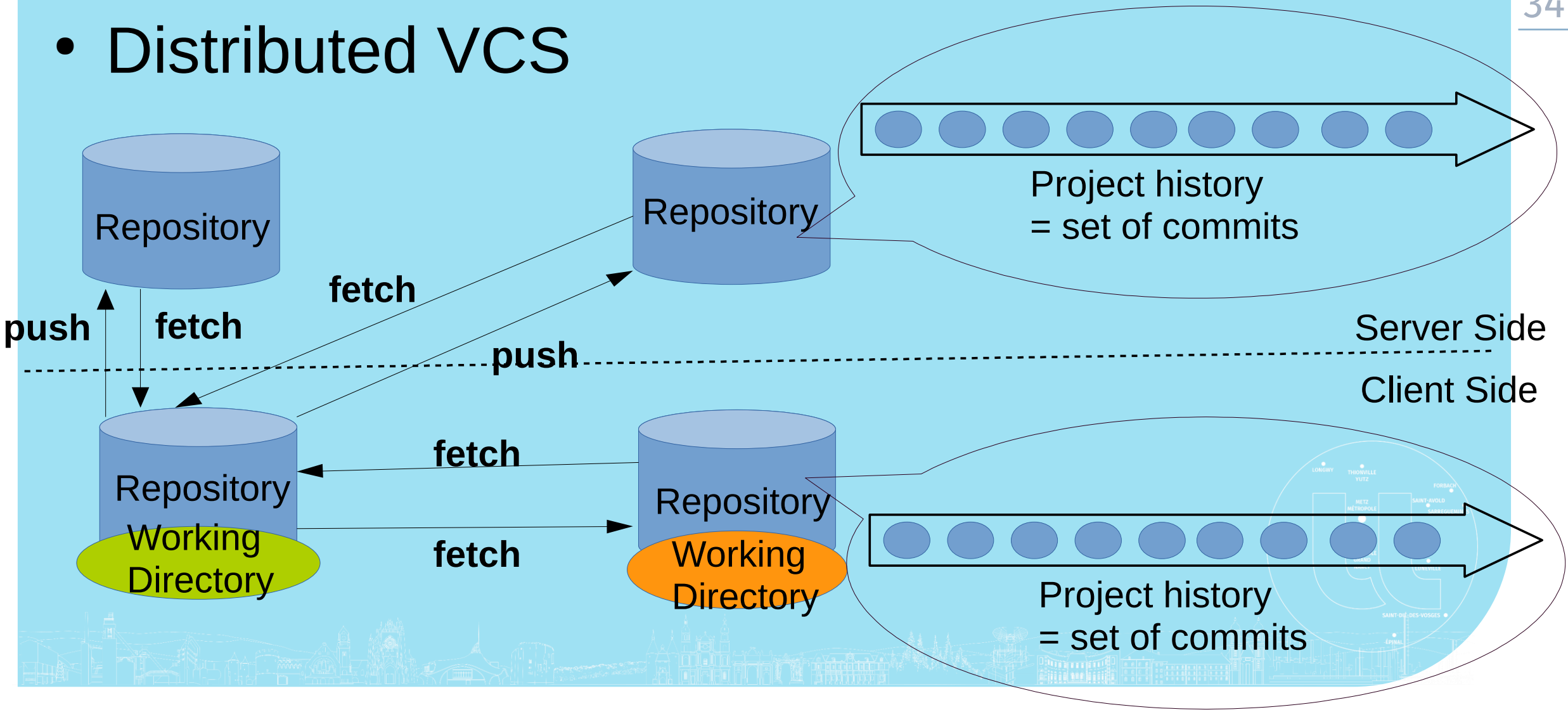
- Based on a peer-to-peer approach to synchronizing operations.
- Repositories exist on both the **server and client** sides.
- On workstations: A working directory is connected to each repository.

• Distributed VCS



! **Fusion Conflicts**

• Distributed VCS



Distributed VCS: Benefits

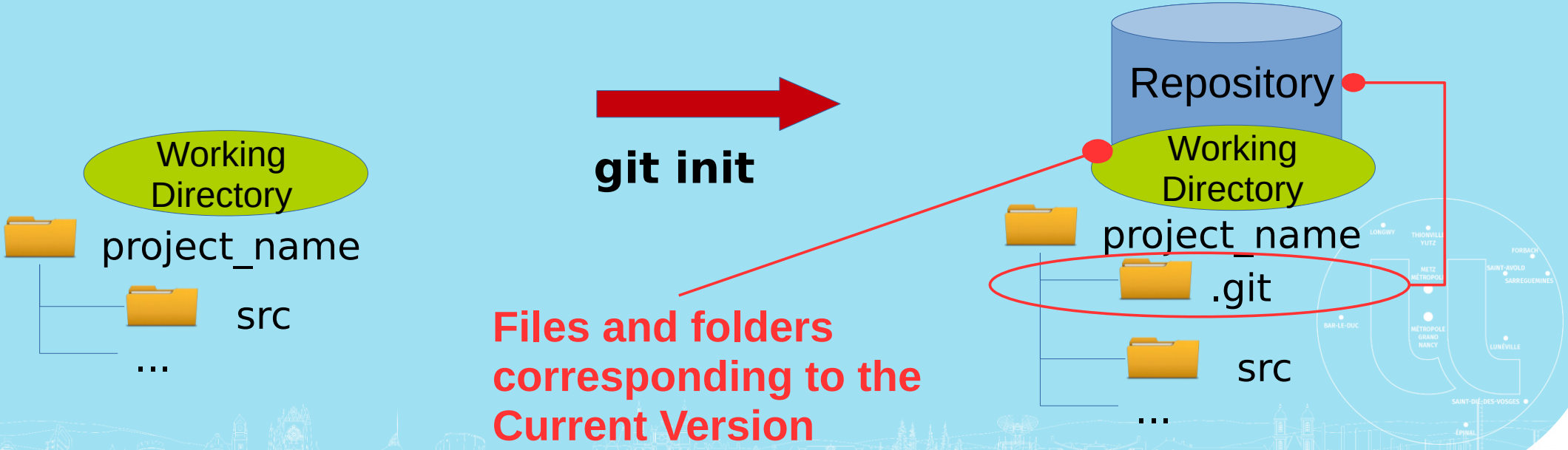
- Robustness: multiple copies of the same deposit.
- Easy offline work: create commits and
- Merging changes is done offline.

Distributed VCS: Limitations

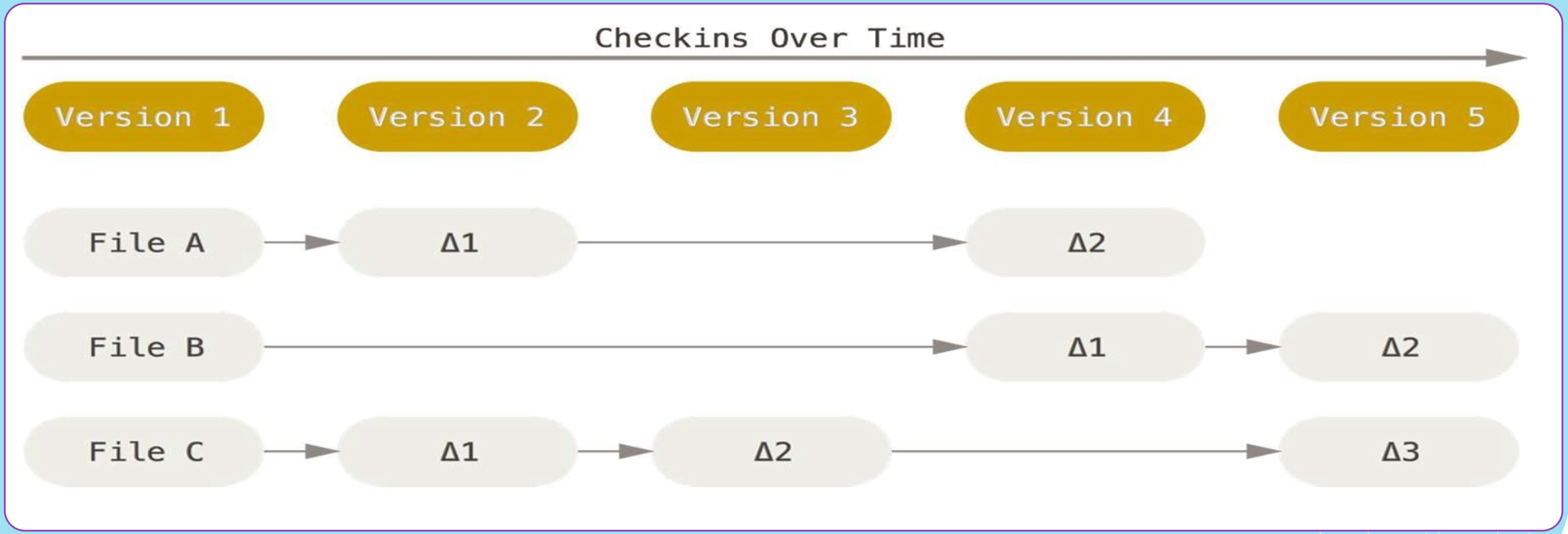
- More commands to understand...
- **Guaranteeing a policy of access to repositories is very difficult with DVCS**
  - Each repository has a full copy of the history (since last **fetch**).
  - Each repository defines its own access rights for its customers.

The **repository**

- A directory whose structure describes the commit graph, branches, and so on.
- Turning a file into a repository: **git init**

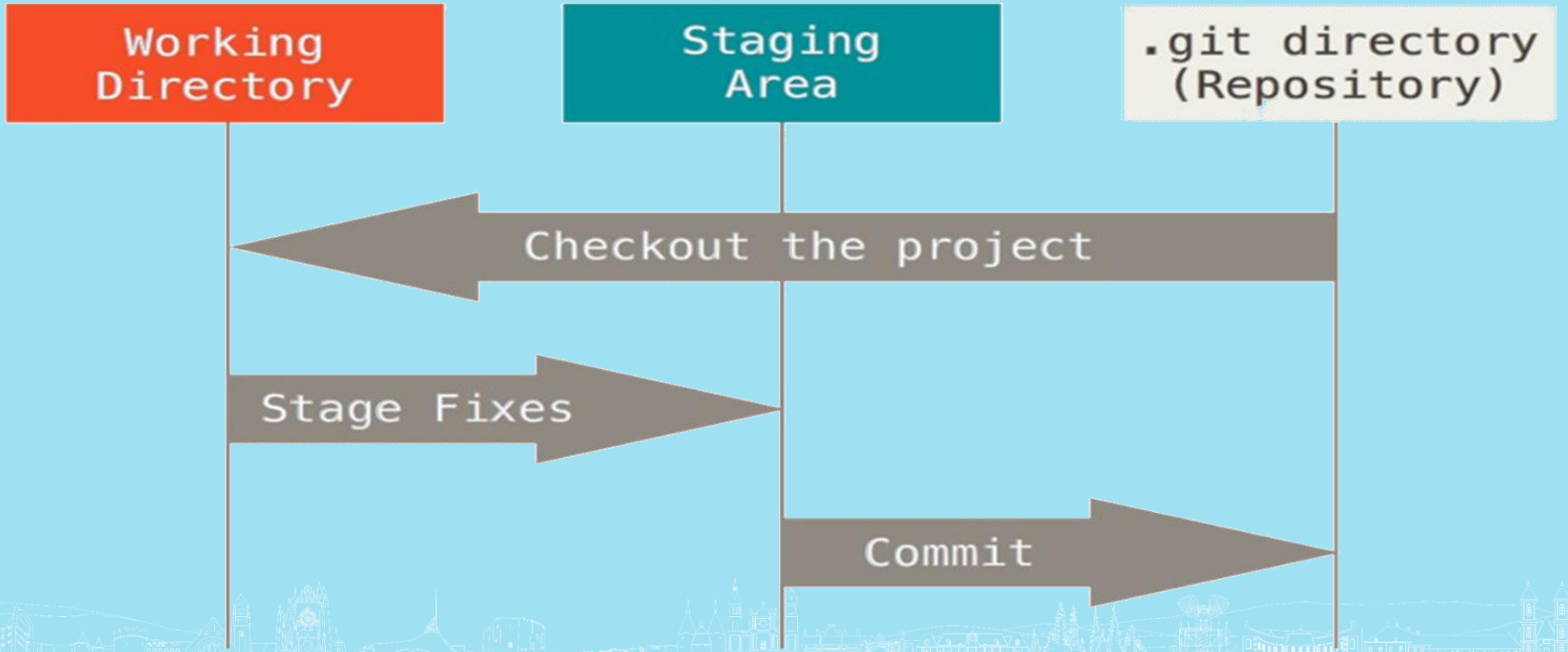


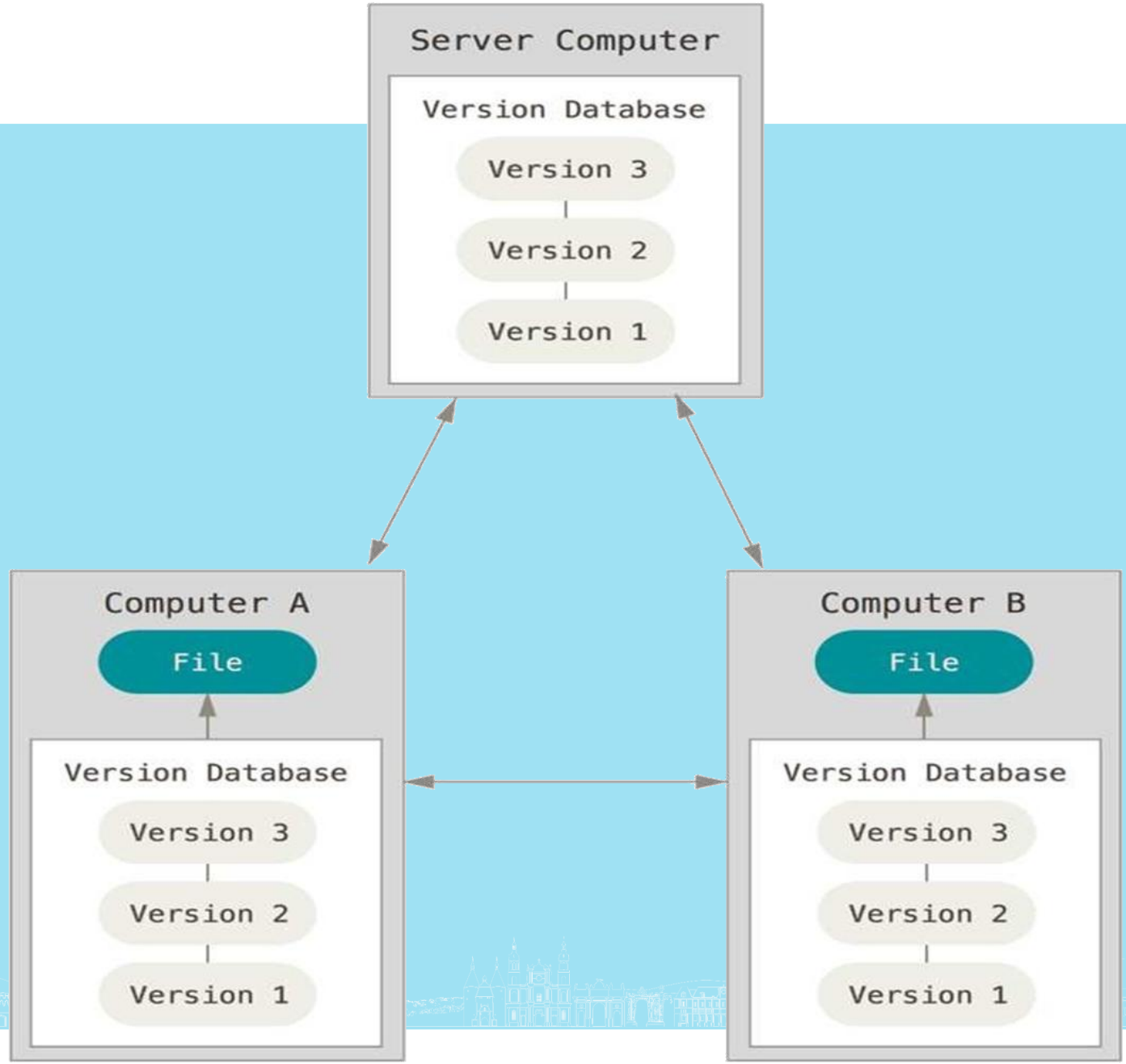
Each time you commit or save the project state, Git takes a snapshot of the contents of your workspace.

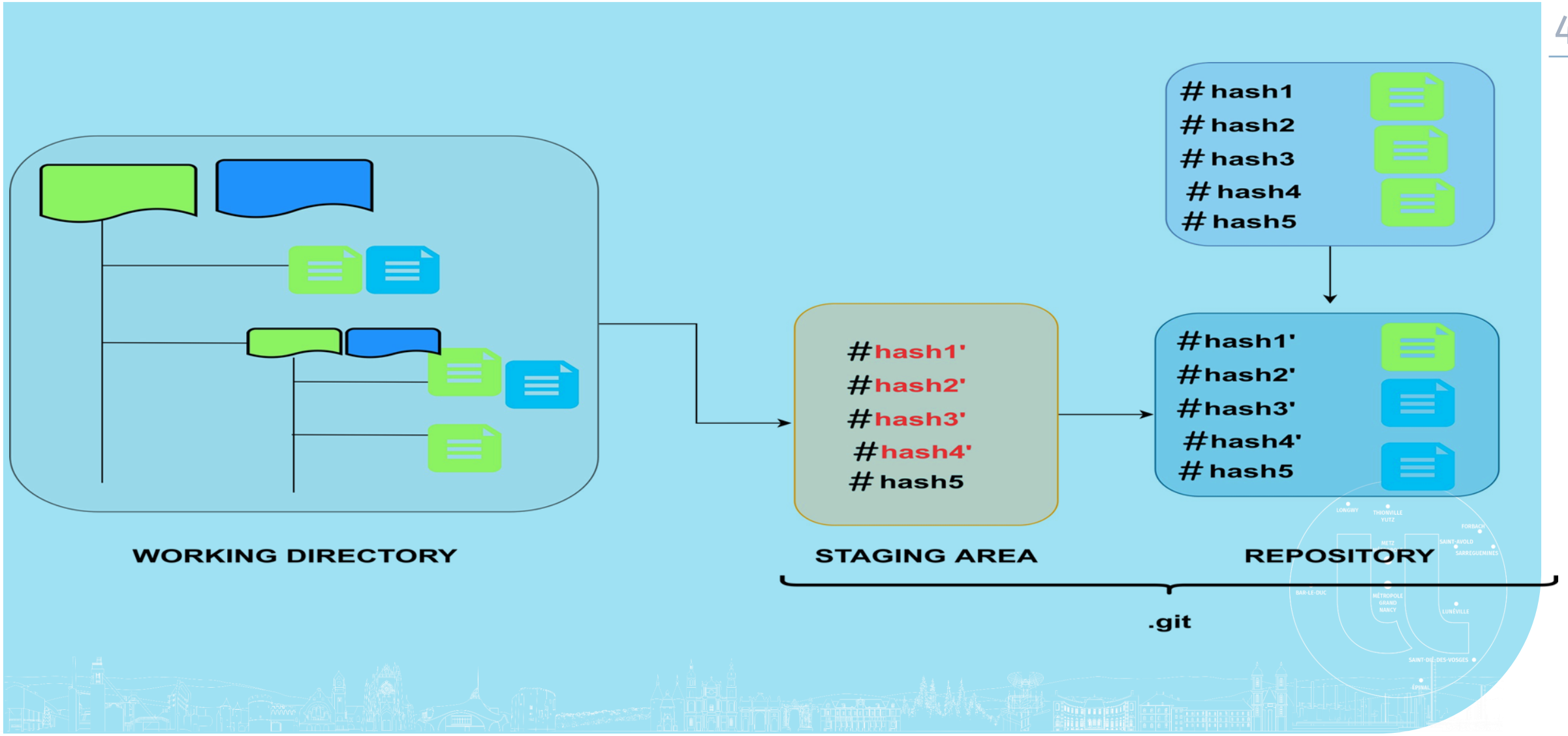


3 main sections of a GIT

- **Validated** = the .git directory
- **Modified** = the current working directory
- **Indexed** = The virtual area that lists the files that will be in the next



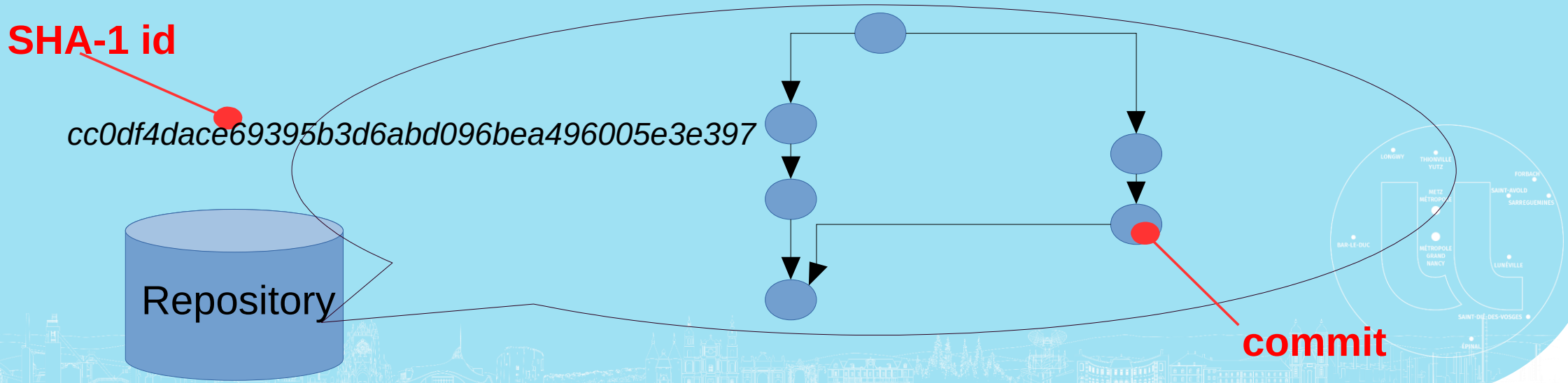

Server Computer
Version Database
Version 3
Version 2
Version 1
Computer A
File
Version Database
Version 3
Version 2
Version 1
Computer B
File
Version Database
Version 3
Version 2
Version 1


#hash1
#hash2
#hash3
#hash4
#hash5
#hash1'
#hash2'
#hash3'
#hash4'
#hash5
#hash1'
#hash2'
#hash3'
#hash4'
#hash5
WORKING DIRECTORY
STAGING AREA
REPOSITORY
.git

The **commit**

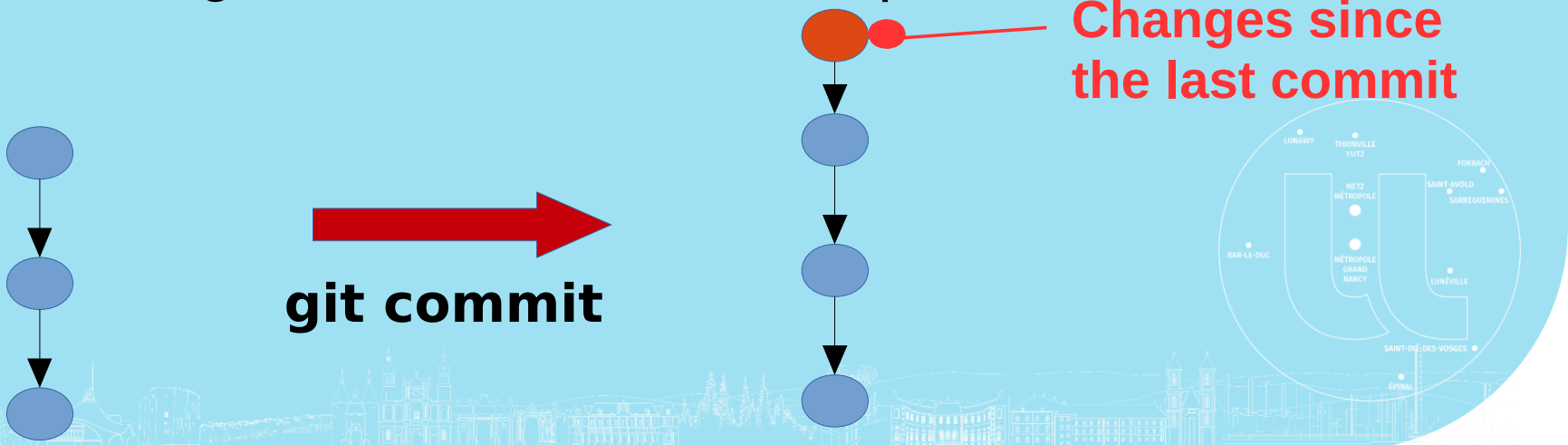
- A set of changes to files and directories.
  - Represented by a **patch** operation.
- Uniquely identified (relative to a project) by a SHA-1 hash code

Commits are structured according to a **directed acyclic graph**



The **commit**

- allows you to commit your changes to create what is called a commit, i.e. a committed step in the code.
- The graph is updated each time a commit is created (e.g. commit action)
  - The new commit sets a pointer to the previous commit and contains the changes to the content of the previous commit.



A commit is composed:

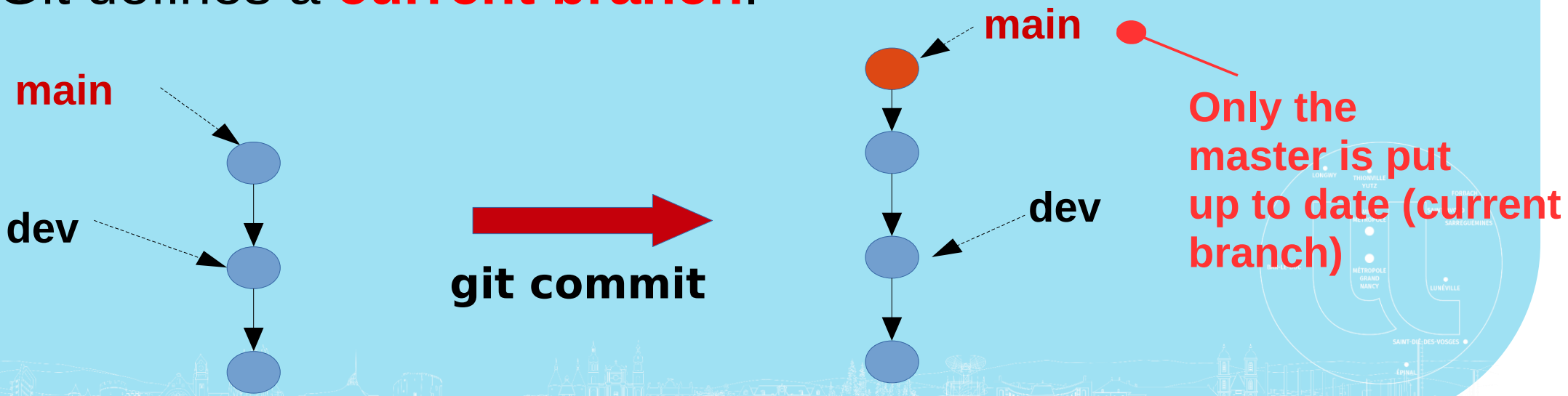
- of a commit message
- a snapshot of the code that is referred to via a unique identifier (a fingerprint or hash)
- of an author
- to a parent commit (or commits) (which allows us to make branches of the Git tree

A commit is composed:

- Ideally, when you make a commit, the code should be in a roughly consistent state and the commit should gather changes that make sense to achieve a goal (e.g., fixing a bug, adding a feature, modifying documentation...)
- Always put a commit message
- Commits are stages in software development. Reading the list of these steps should help a developer understand how the code is evolving.
- a commit is always a reference to a specific version of the entire code in relation to the Git tree, it's not just additions and deletions compared to the code of the previous commit

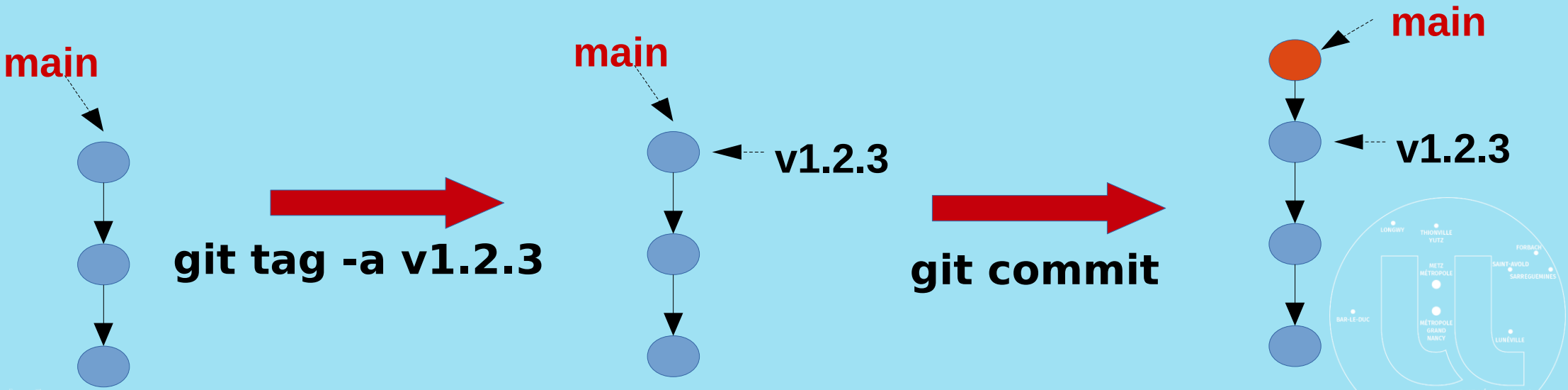
The **branch**

- A **pointer** to (i.e. a reference to) has a commit.
- Mis à jour chaque fois qu'un commit est créé, pointe alors vers ce nouveau commit.
- Default branch: **main (master)**.
- Git defines a **current branch**.



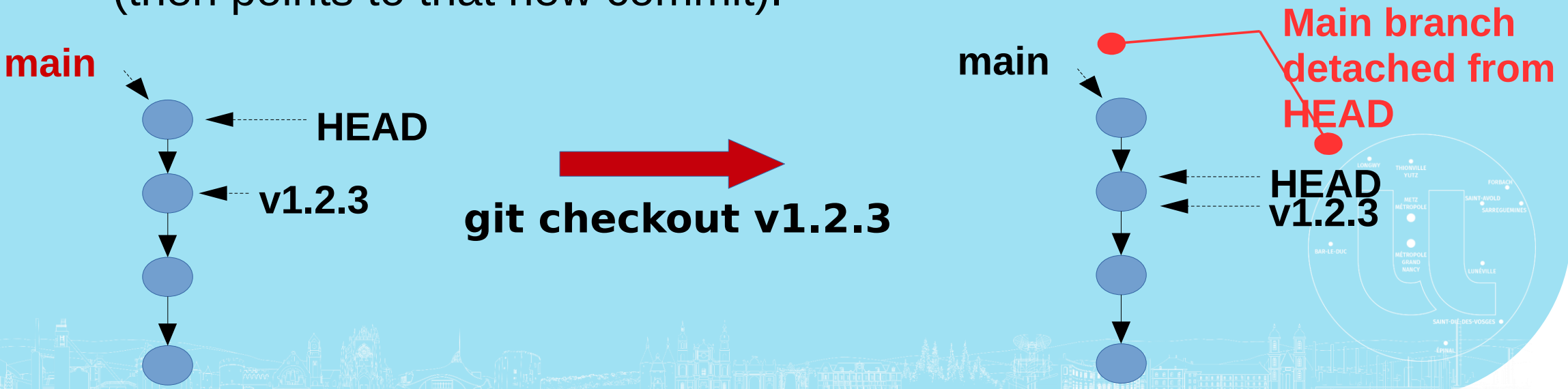
The **tag**

- A **pointer** to (i.e. a reference to) has a commit.
- Allows you to memorize interesting versions.



The **HEAD pointer**

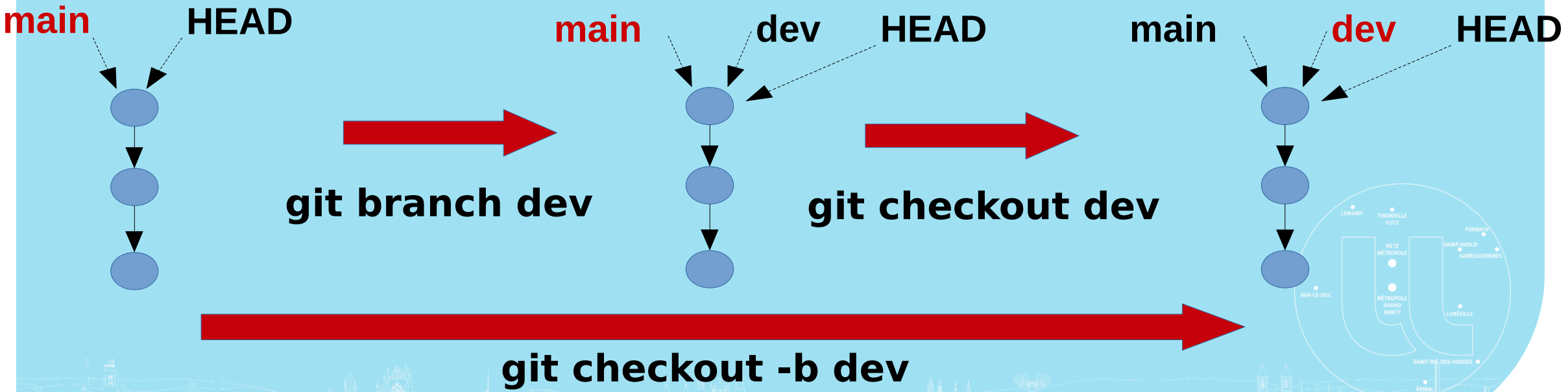
- A **pointer** to (i.e. a reference to) a commit.
- Represents the **current state of the working directory**: changes applied to the project up to the targeted commit.
- Allows you to navigate through the graph to find a given version.
- Updated on command or automatically each time a **commit** is created (then points to that new commit).



Créer des **branches**

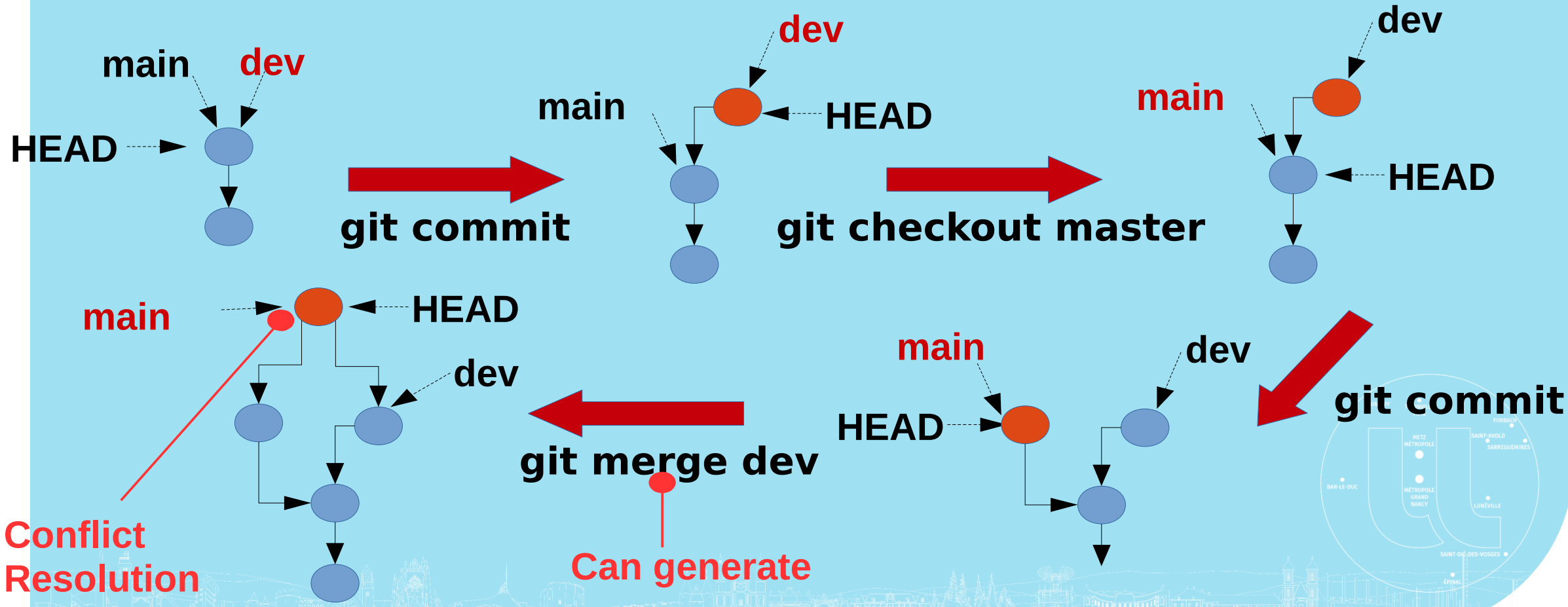
- Plusieurs branches dans le même dépôt.

HEAD.



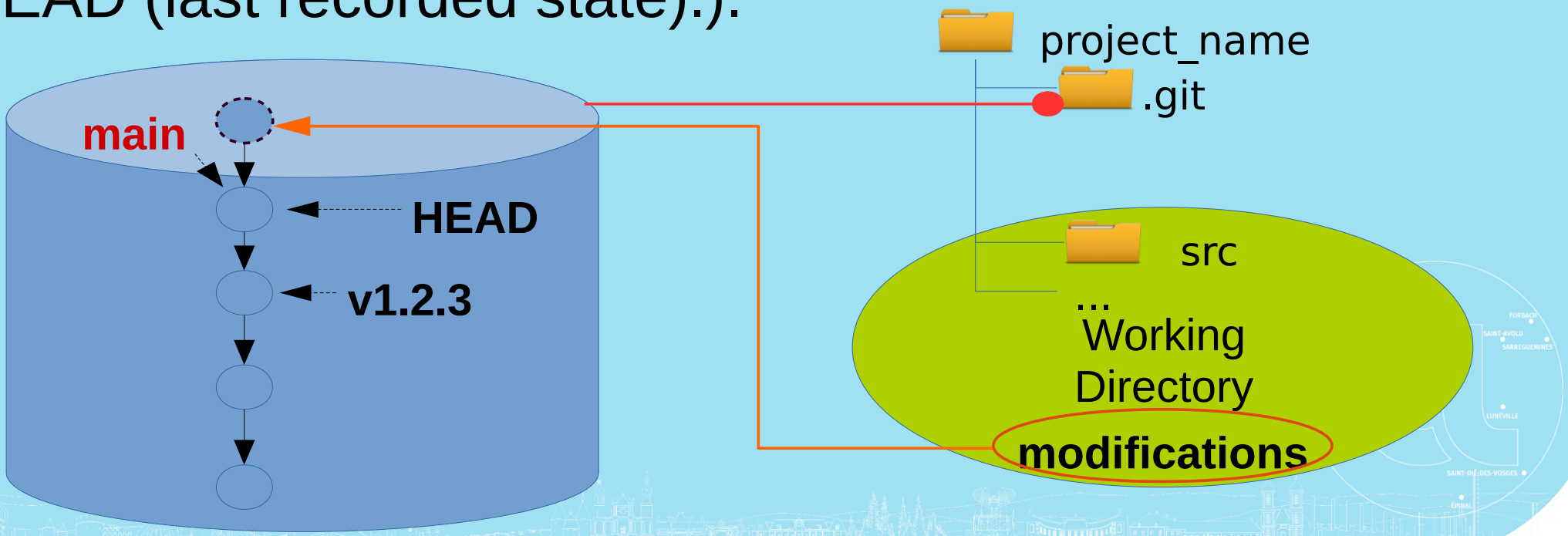
UNIVERSITÉ DE LORRAINE

• Fusion (*Merge*) of branches



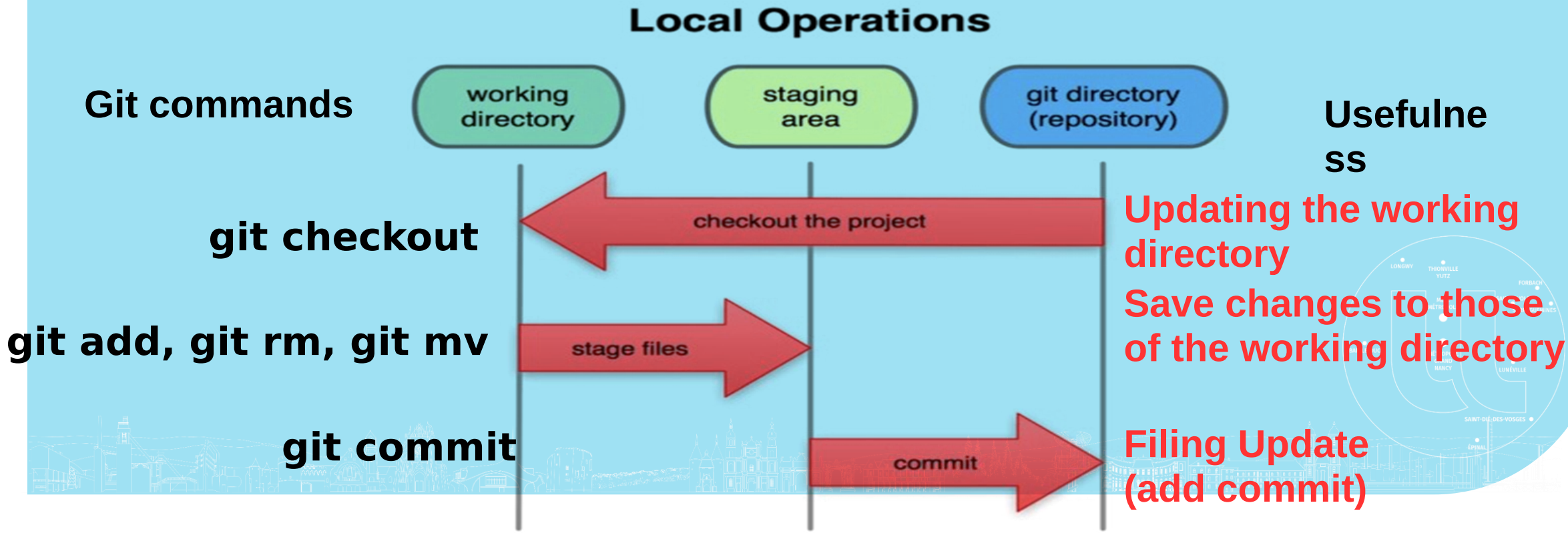
Create a commit from changes

- Goal: save changes between the contents of the working directory (folders and files) and the commit pointed out by HEAD (last recorded state).).



Create a commit from changes

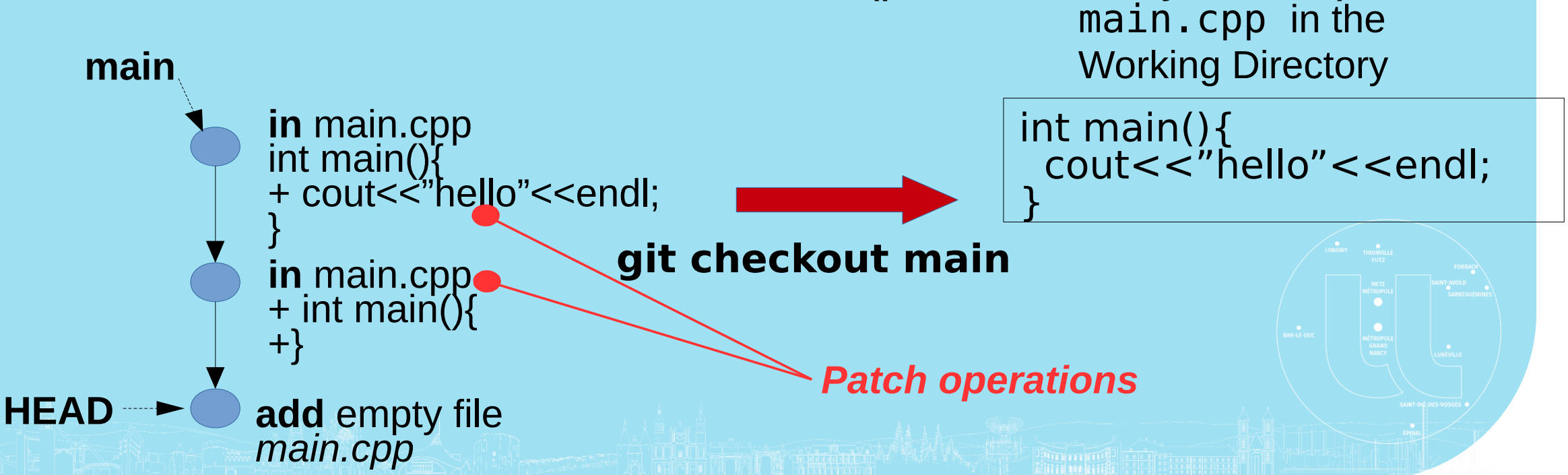
- Originality of git: 2 phases



UNIVERSITÉ DE LORRAINE

Updating the working directory

- At each checkout or merge.
- Applies the sequence of changes (patches) and then the first one until the current commit (pointed out by **HEAD**).



Saving changes to the working directory

- Sélectionner les modifications qui seront contenu dans le prochain commit.
  - internal files (use **git add –p**); All changes to a file or new file (use **git add filename**)
  - All changes (use git add –all)
- Deselecting changes is possible (usegit reset)

main.cpp in **HEAD** in the **repository**

```
int main(){
  cout<<”hello”<<endl;
}
```

main.cpp in the **working directory**

```
int main(){
  cout<<”hello”<<endl;
  return 0;
}
```

git add main.cpp

Saved Change in the **staging area** **in**

```
main.cpp
cout<<”hello”<<endl;
+ return 0;
}
```

Diff operation to deduce the patch.

Updating the local repository (“commit”)

- Create a new commit from the changes saved in the « *staging area* ».
- Added a message explaining what has been changed.

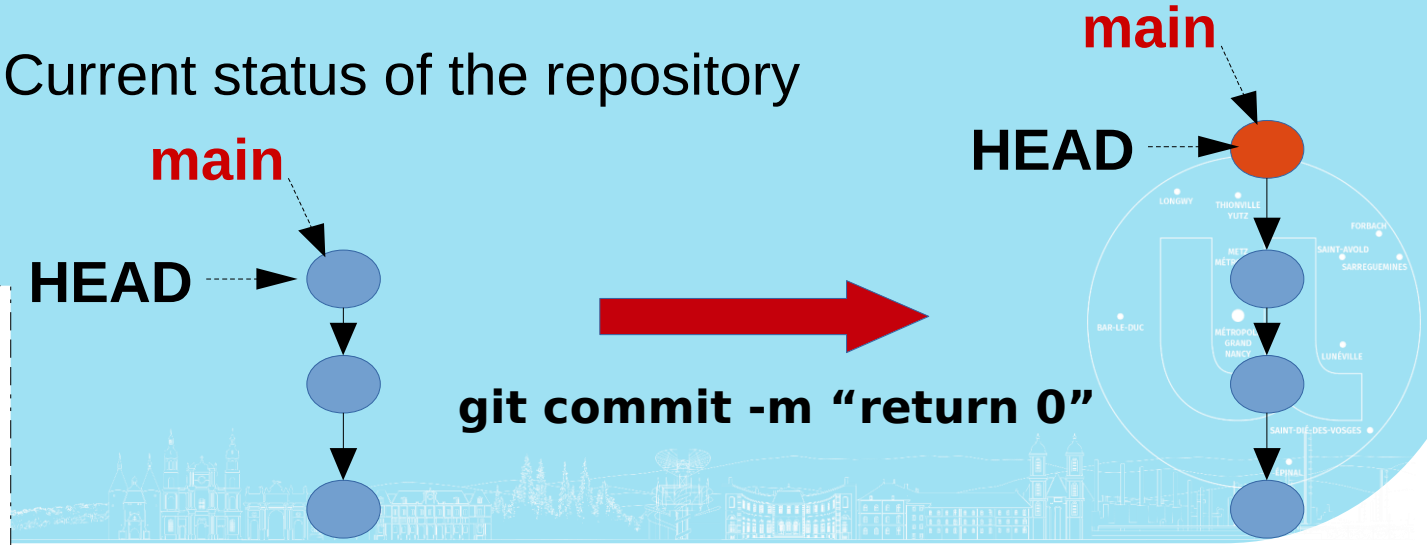
Saved changes in the **staging area**

add other.cpp

in main.cpp

```
 in main.cpp
 cout<<”hello”<<endl;
+ return 0;
}+ return 0;
 }
```

Current status of the repository

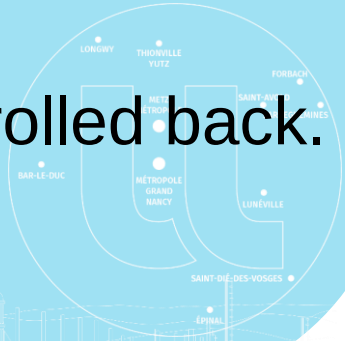


Why 2 phases to create commits?

- Create "small commits" from a large number of changes.
- Delay the commit of certain changes: Committed features can be "committed" immediately.
- isolate code that will never be committed (e.g. debug traces).

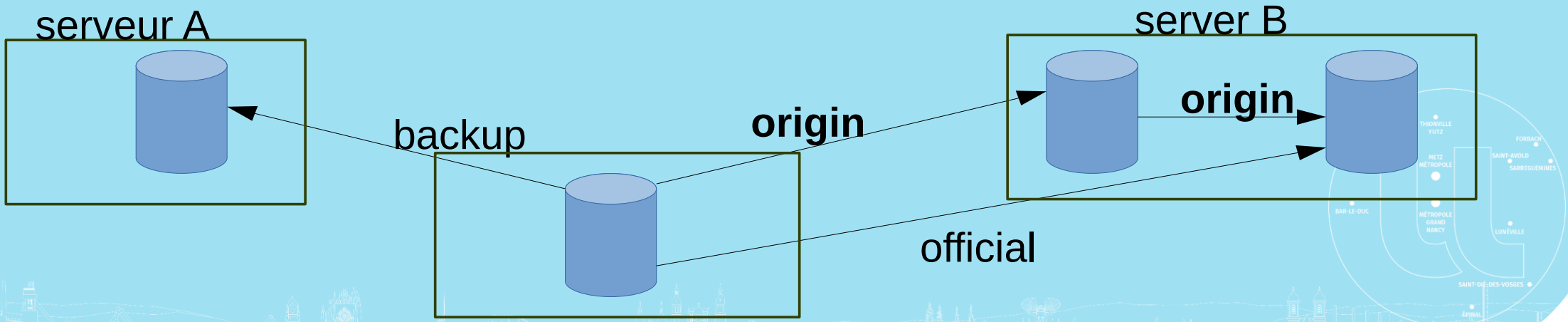
Once the interesting changes are "committed", cleaning up the staging area and working directory is possible:

- Use **git reset –hard** : permanent deletion of uncommitted changes.
- Use **git stash save** : uncommitted changes can be rolled back.



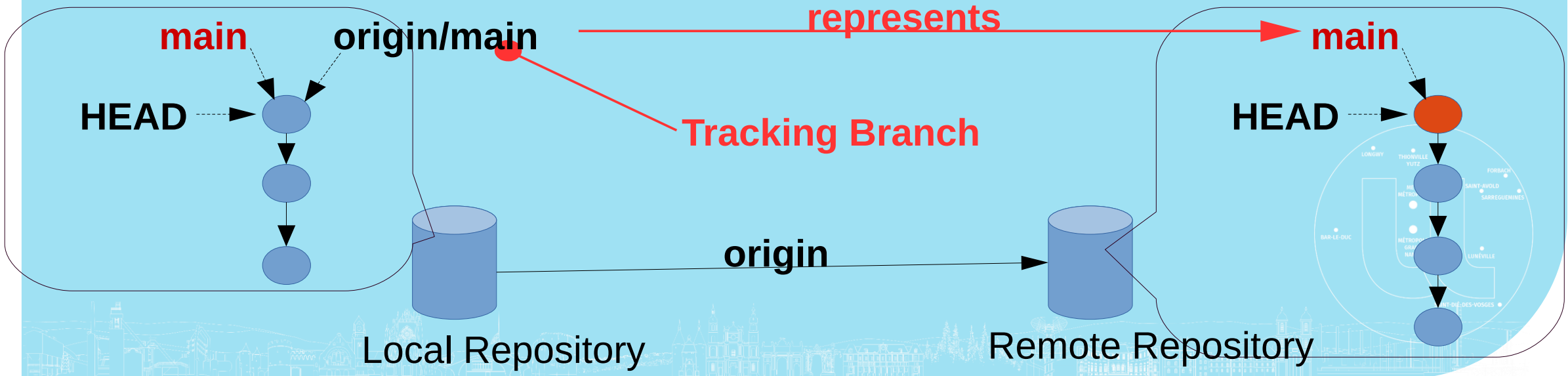
Synchronization between remote repositories

- Each repository has a set of repositories called ***remotes***.
- The default *remote* is called **origin**.
- Each remote is associated with a **name** (unique in the local repository) and defines a **url** (address of a remote repository).).
- Add a *remote* :**git remote add backup <address>**
- Delete a *remote* :**git remote rm backup <address>**



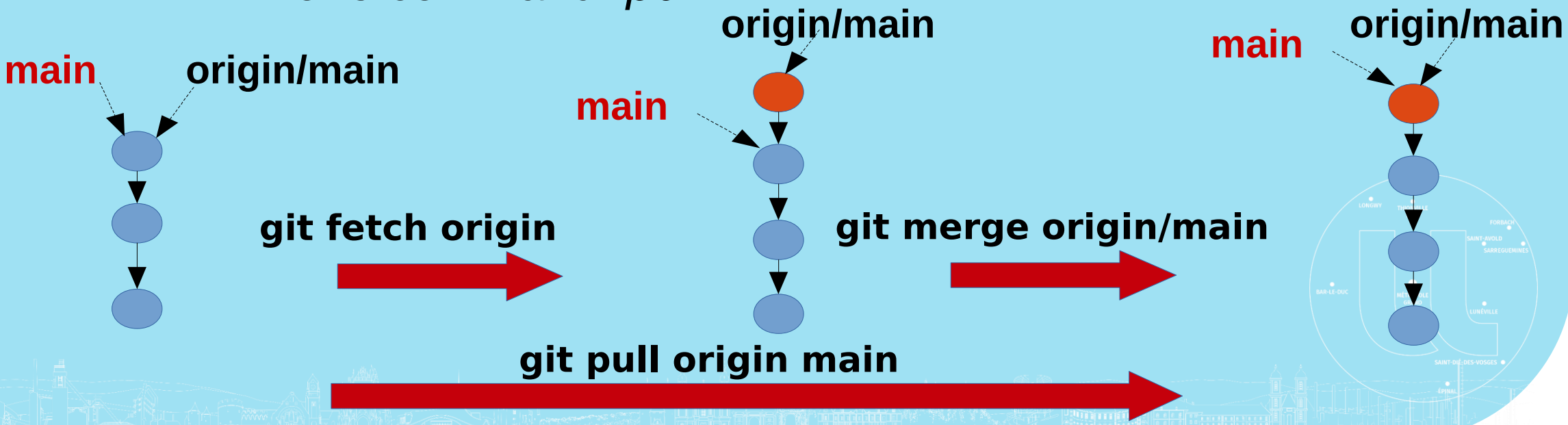
Synchronize **local** branches with **remote** branches

- The branches of the remotes are known in the local repository but they are **read-only** (cannot create commits on these branches). These are called **tracking branches** (***remote tracking branches***).



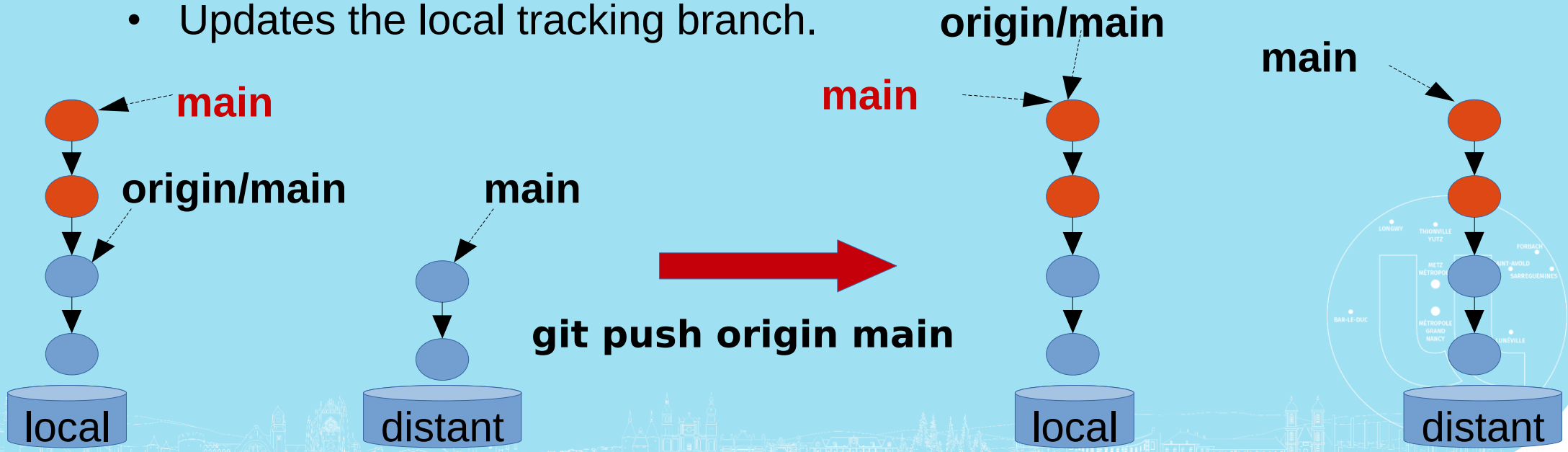
Updating a repository from one of its *remotes*.

- Update (if necessary) the **local commit graph** and **tracking branches** (*fetch* command).
- Local branches are **merged** with tracking branches (merge command).
- All-in-one command: *pull*



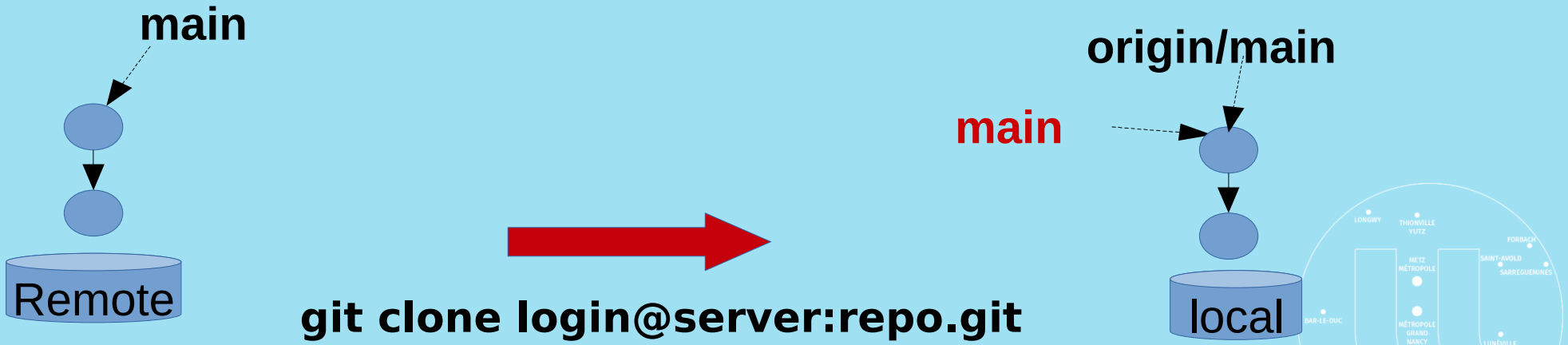
Update a remote branch from a local branch

- Use the command ***push*** (opération atomique)
  - Verifies that the corresponding local tracking branch is up to date.
  - Updates the remote repository's commit graph and branch
  - (of the same name).
  - Updates the local tracking branch.



Clone, the most well-known git operation corresponds to:

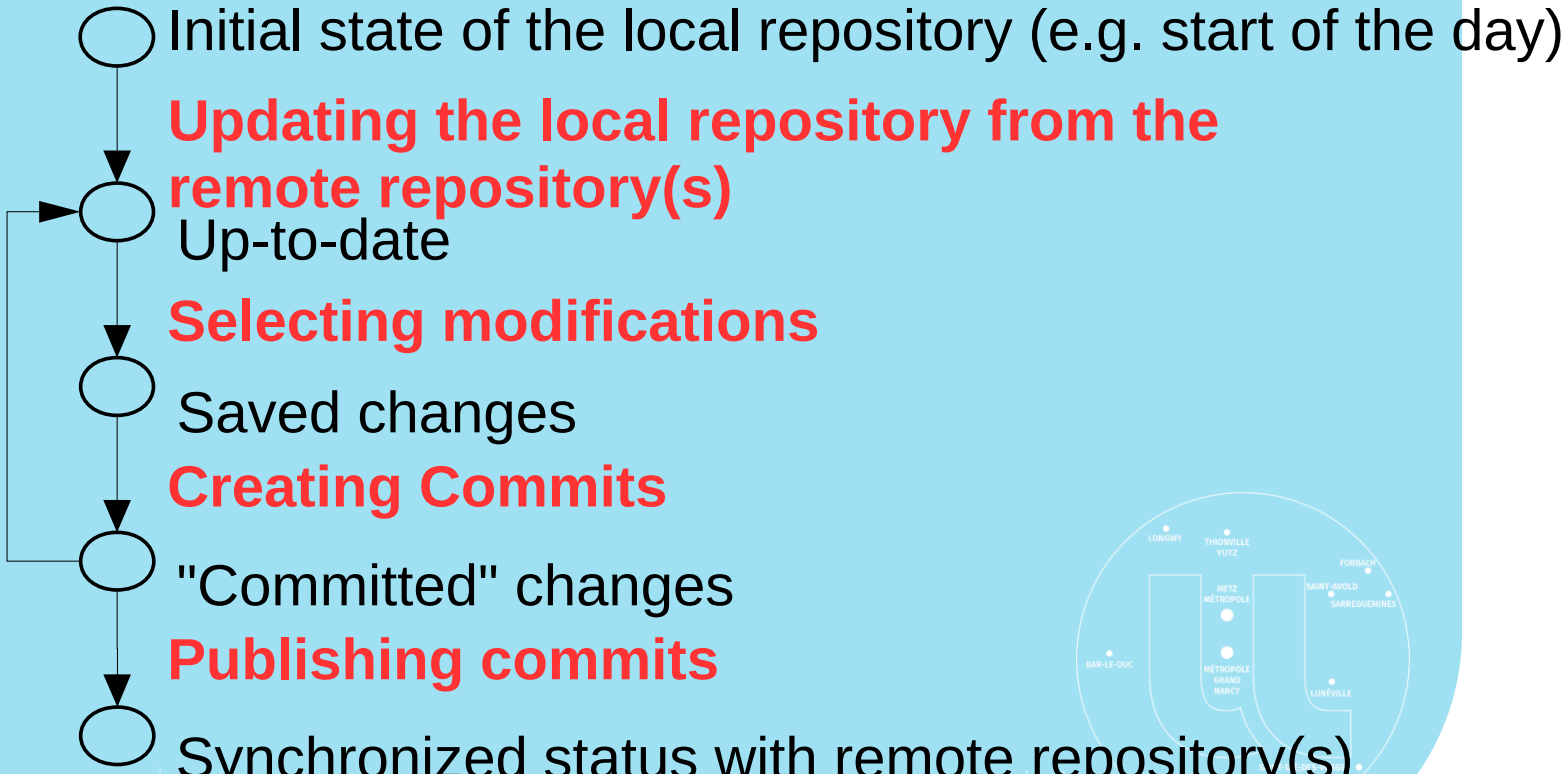
- Create a local directory and initialize it as a repository (**git init**).
- Create a *remote* called **origin** (**git remote add origin <address>**)
- Create a local branch ***main*** (**git checkout -b main**)
- Local branch Update ***main*** (**git pull origin main**)



Adresse: login@server:repo.git

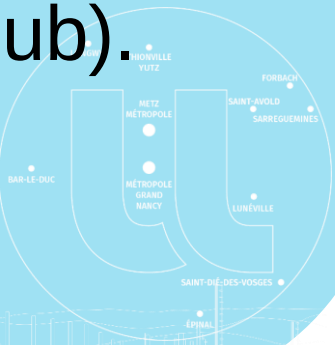
- Typical use

```
git pull/fetch/merge ...

git add/rm/mv/reset ...

git commit ...

git push
...
```



Web application to manage **server-side** repositories

- Creation/deletion of repositories.
- Registration and management of **access rights** to repositories.
- Graphical tools for viewing and editing: graph of commits, branches, tags, files, etc.
- Project documentation (README files)
- Project communication tools (**issues**)
- **Fork** mechanism and **merge requests** (like on github).



Gitlab server: https://www.gitlab.com

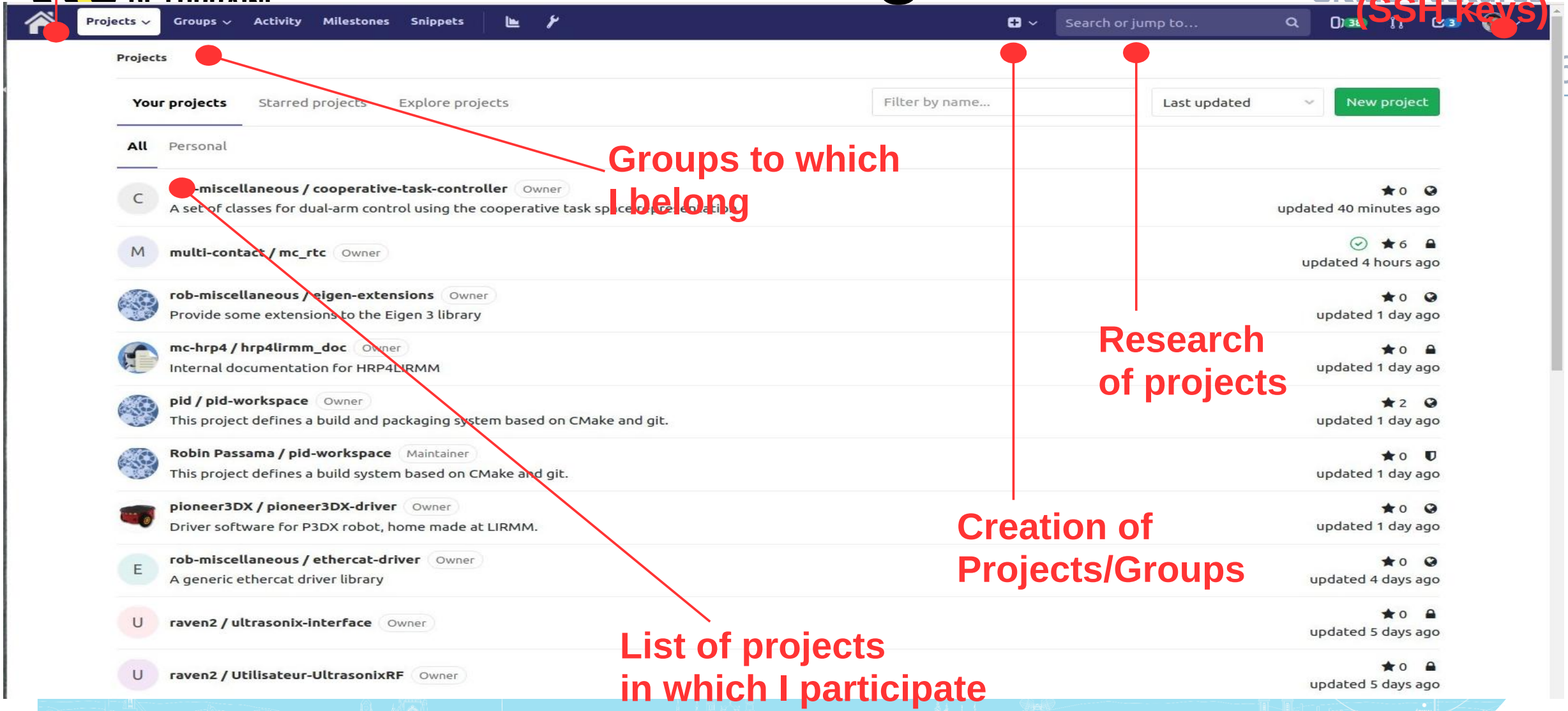
Definitions

- *Project*: git repository + other information (access rights, wiki, issues, continuous integration, etc.).
- *Workspace*: A **directory** containing git repositories.
- *User*: The person registered on the server.
- Each user has a **personal workspace** (apart from external users).
- *Group*: A group of users working on a set of common projects. Each group has a **workspace** containing the repositories of these projects.

Home Page

Direct access to this page

Settings
Users
Gitlab Features
(SSH keys)



Groups to which I belong

Research of projects

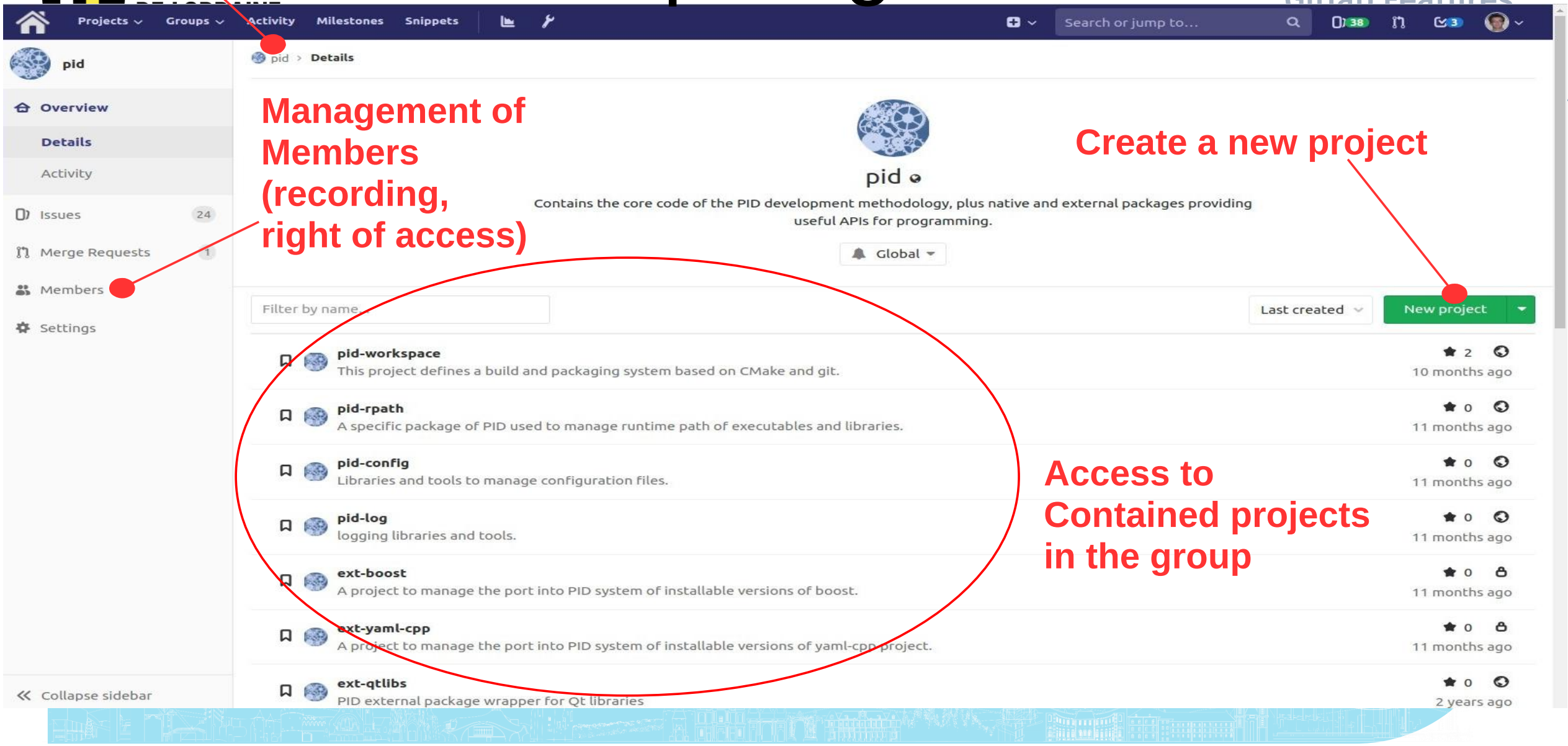
Creation of Projects/Groups

List of projects in which I participate

Group Page

Group Workspace

Gitlab Features

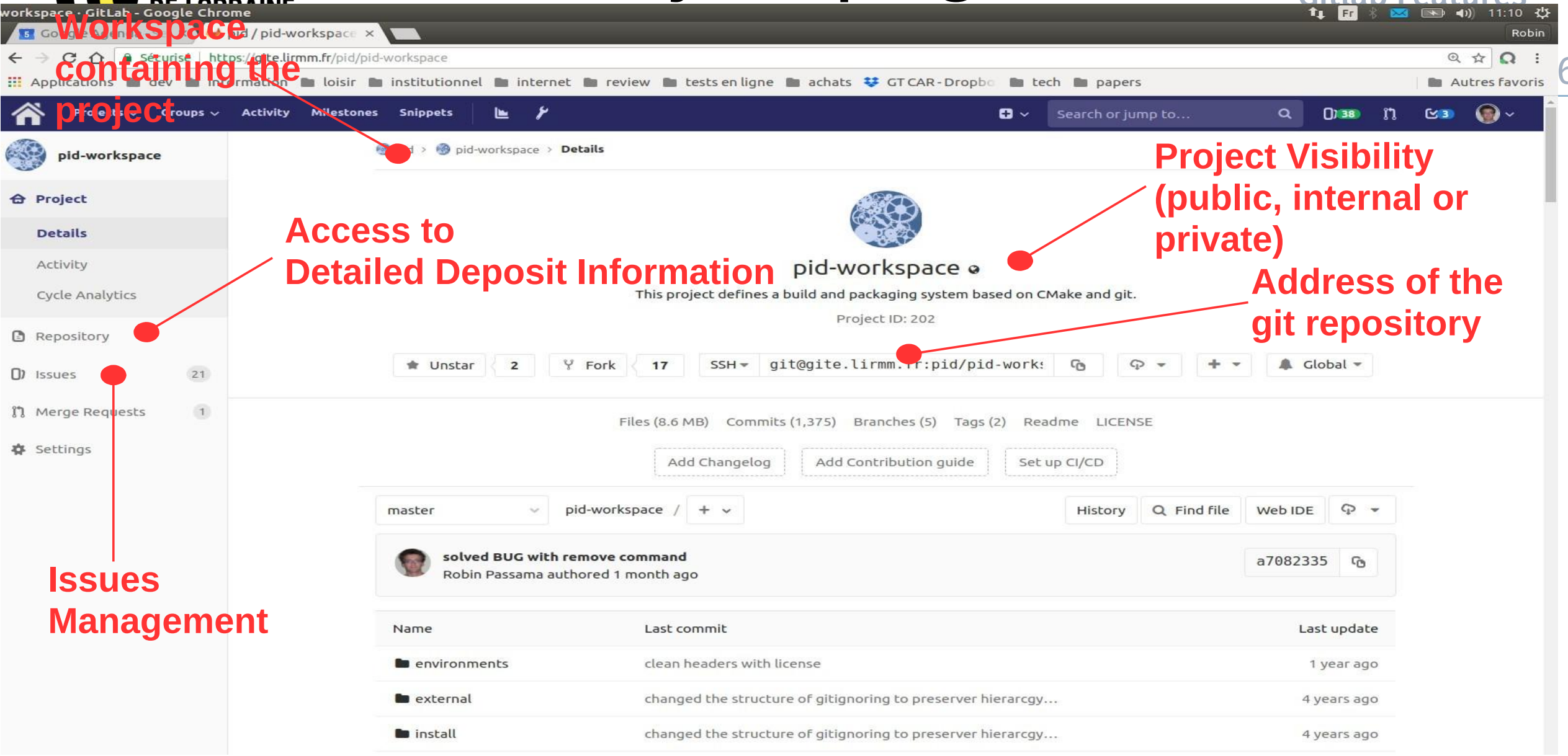


Management of Members (recording, right of access)

Create a new project

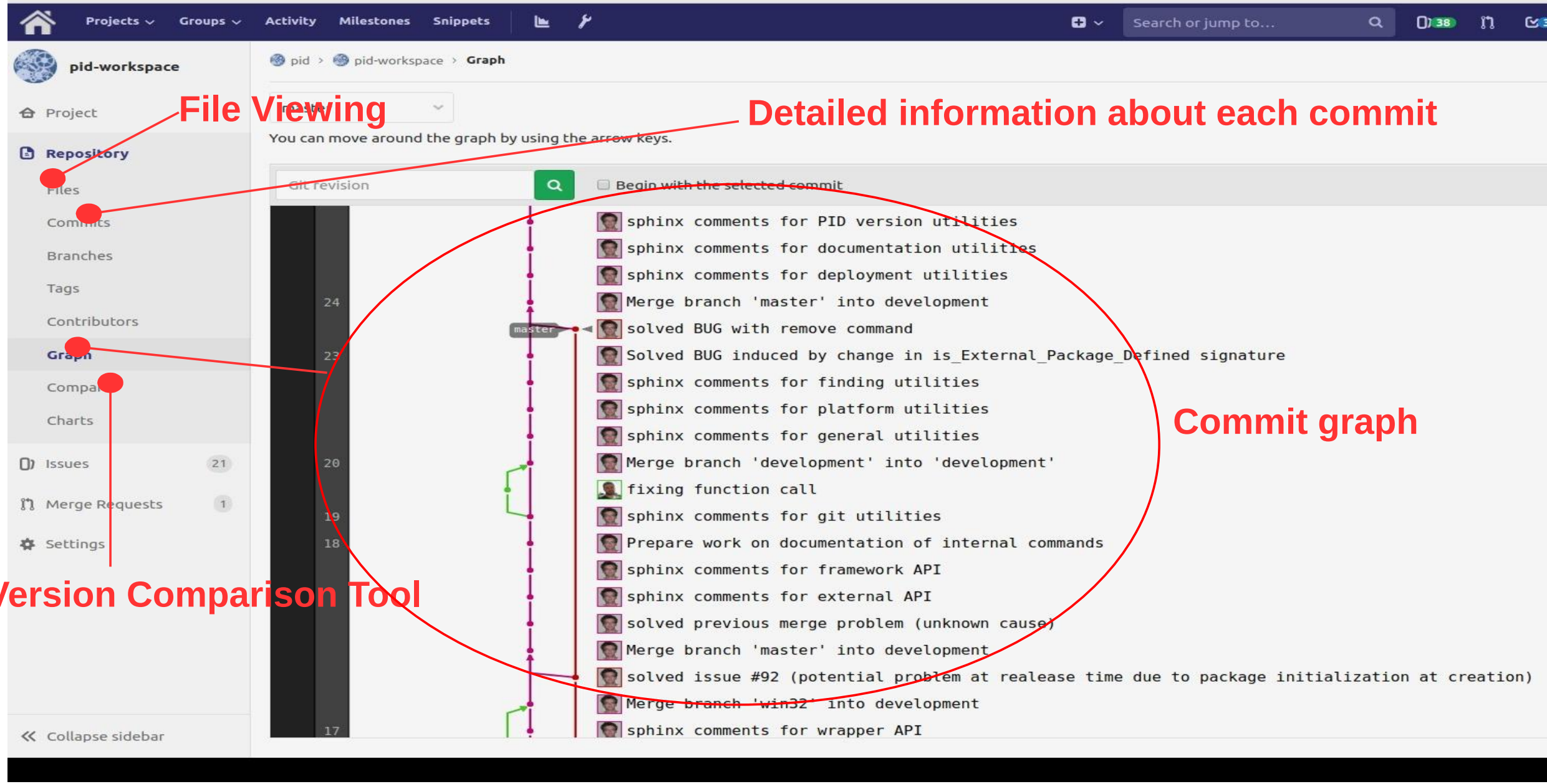
Access to Contained projects in the group

Project page



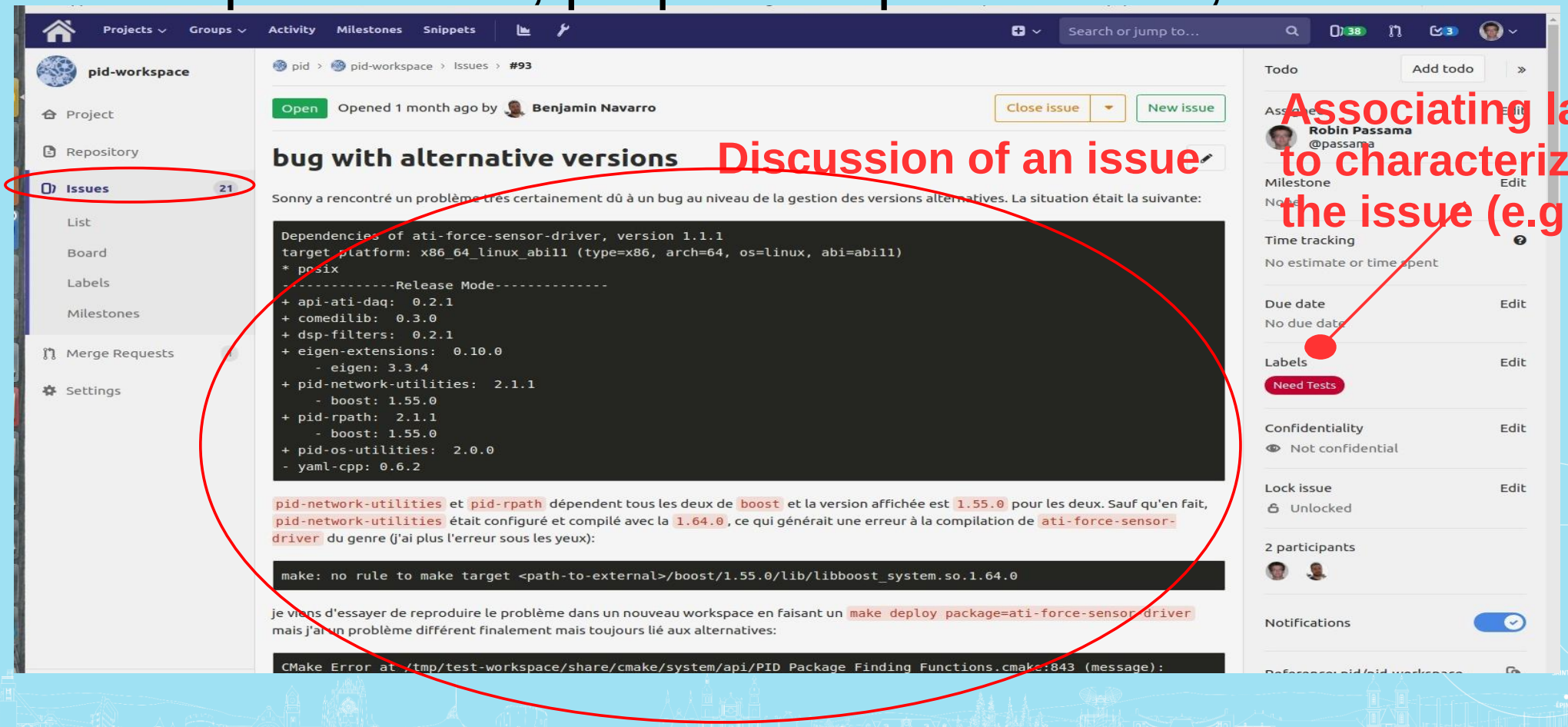
Deposit Information

Gitlab Features

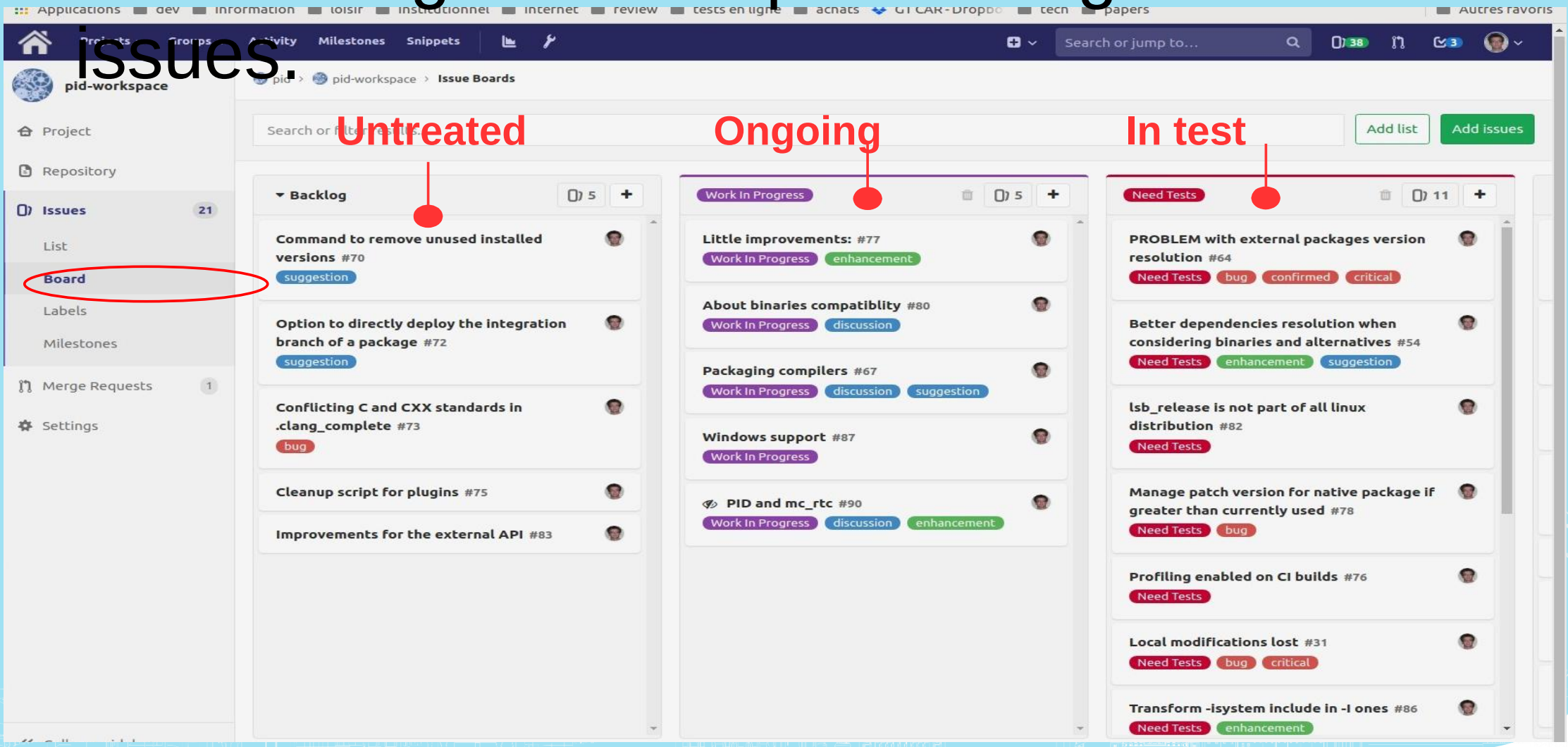


Developer/User Discussion

- Report BUGS, propose improvements, etc.



- *Board* : Organize the processing of issues.

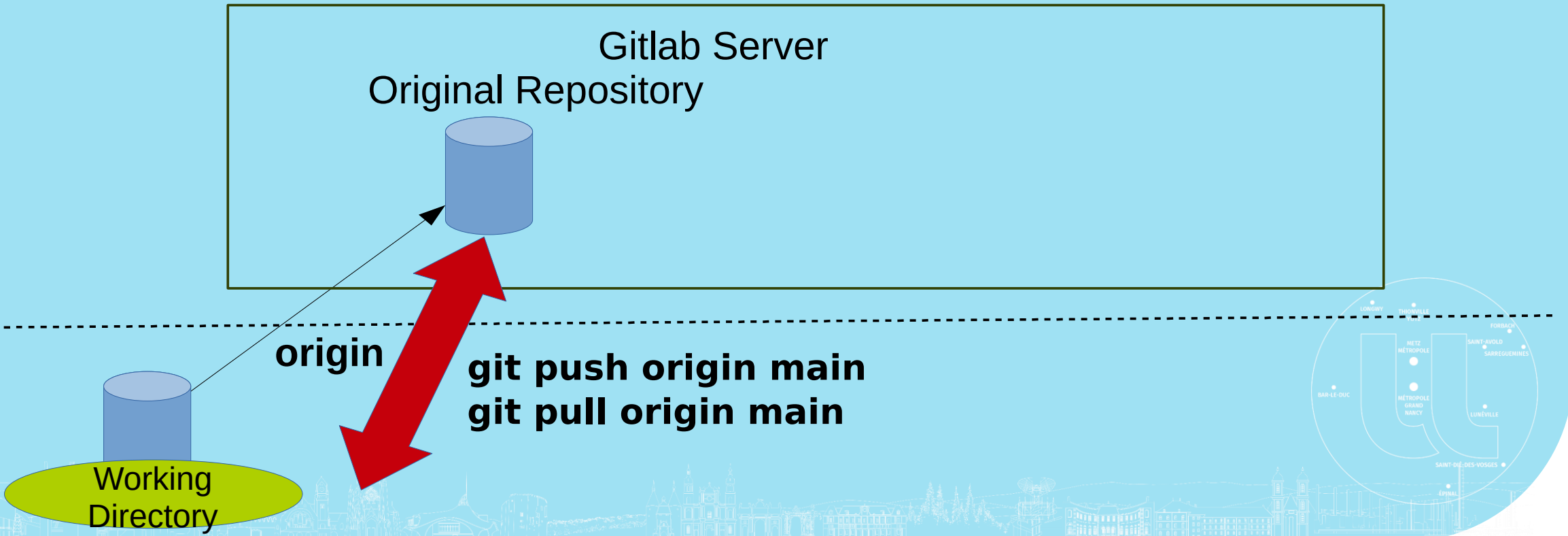


Github-like Work-flow:

- *Fork* = clone a repository directly in Gitlab (the created repository exists in the server)
  - The **original** repository "knows" its **cloned** repositories.
- Merge request = propose to merge a branch of the **cloned** repository into a branch of the **original** repository.
  - The request can be discussed, approved or rejected.
  - Opportunity to see proposed changes.
  - Only the developers of the original repository can validate the request.

Allows you to receive contributions from people outside the project

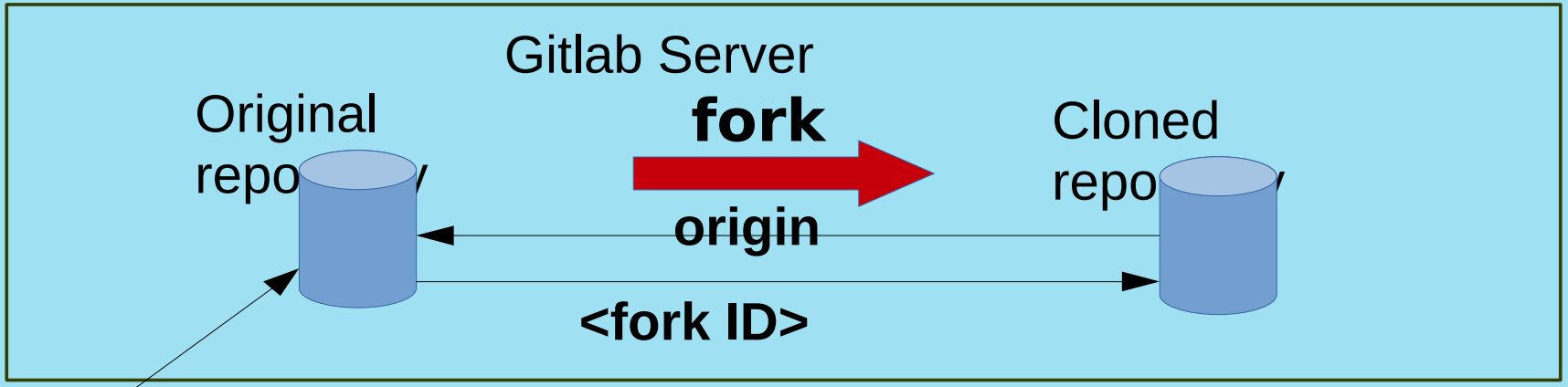
Initial situation

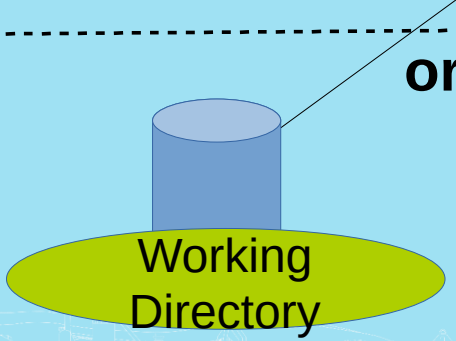


UNIVERSITÉ DE LORRAINE

fork

Clone one repository into another *workspace*



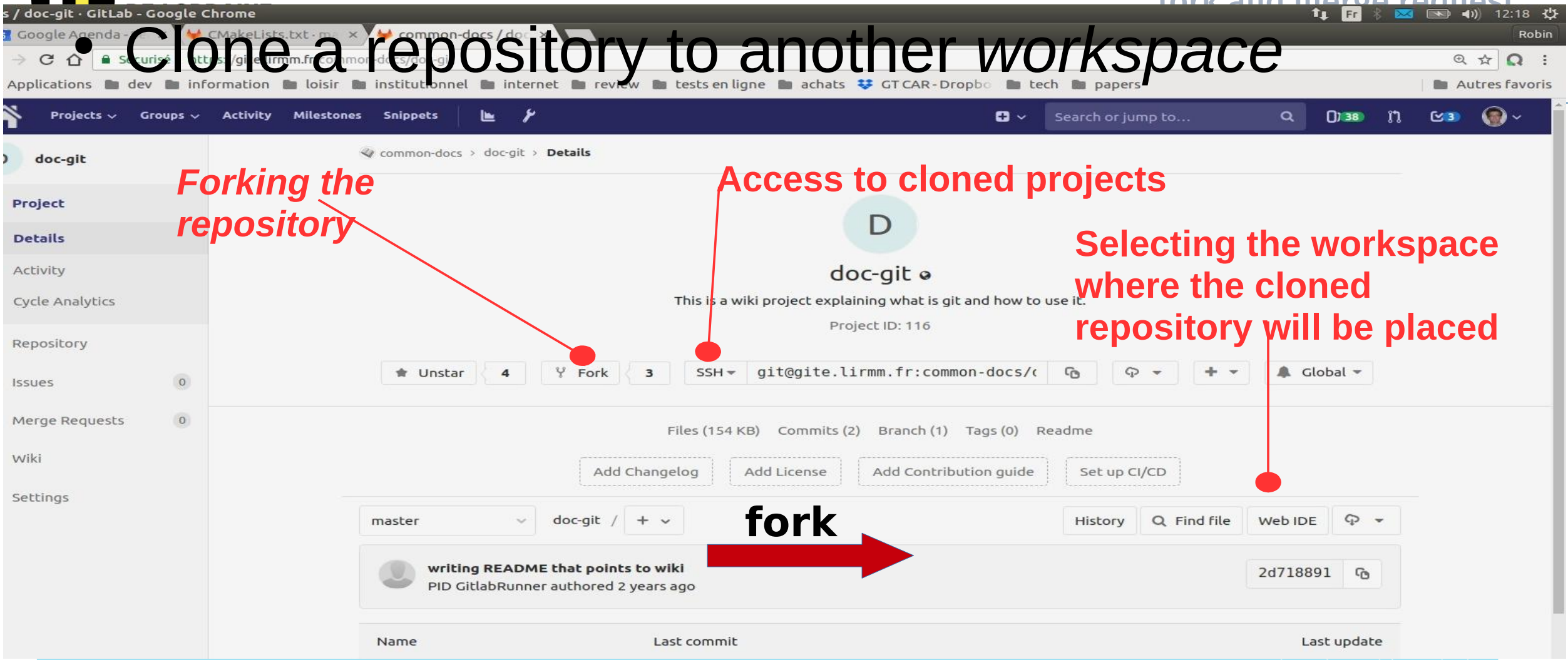


"Official" developers

UNIVERSITÉ

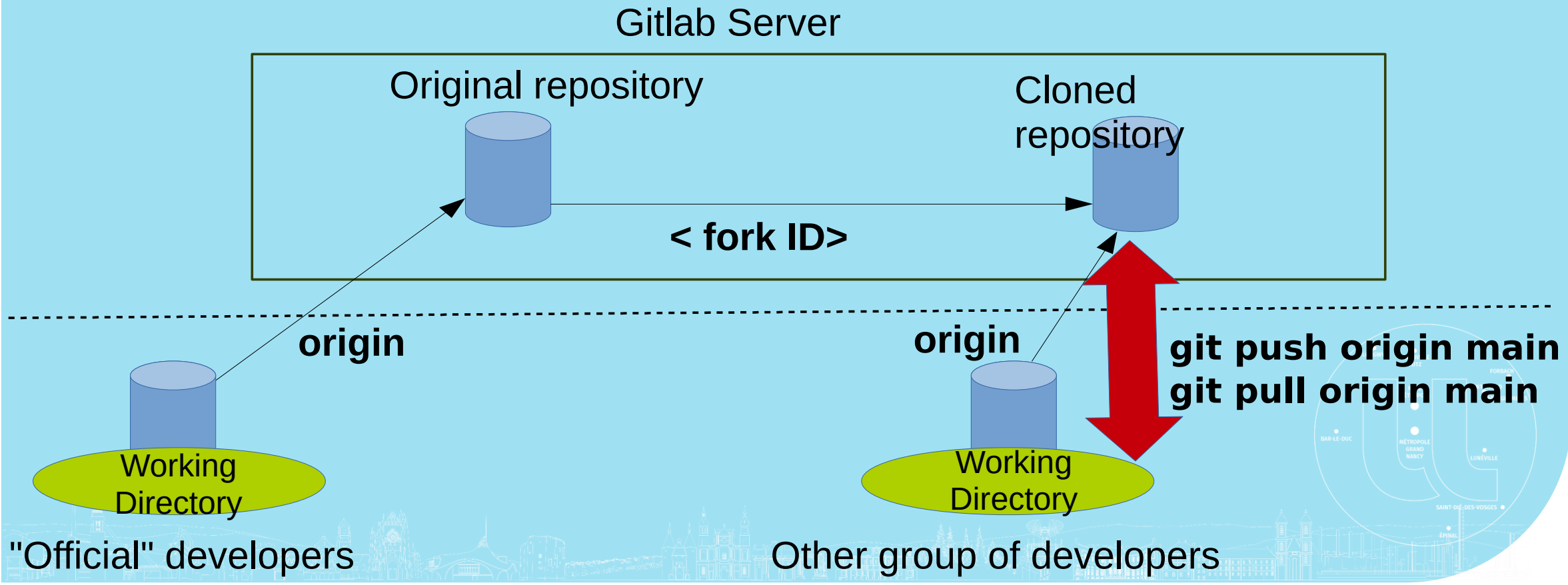
forke

- Clone a repository to another *workspace*



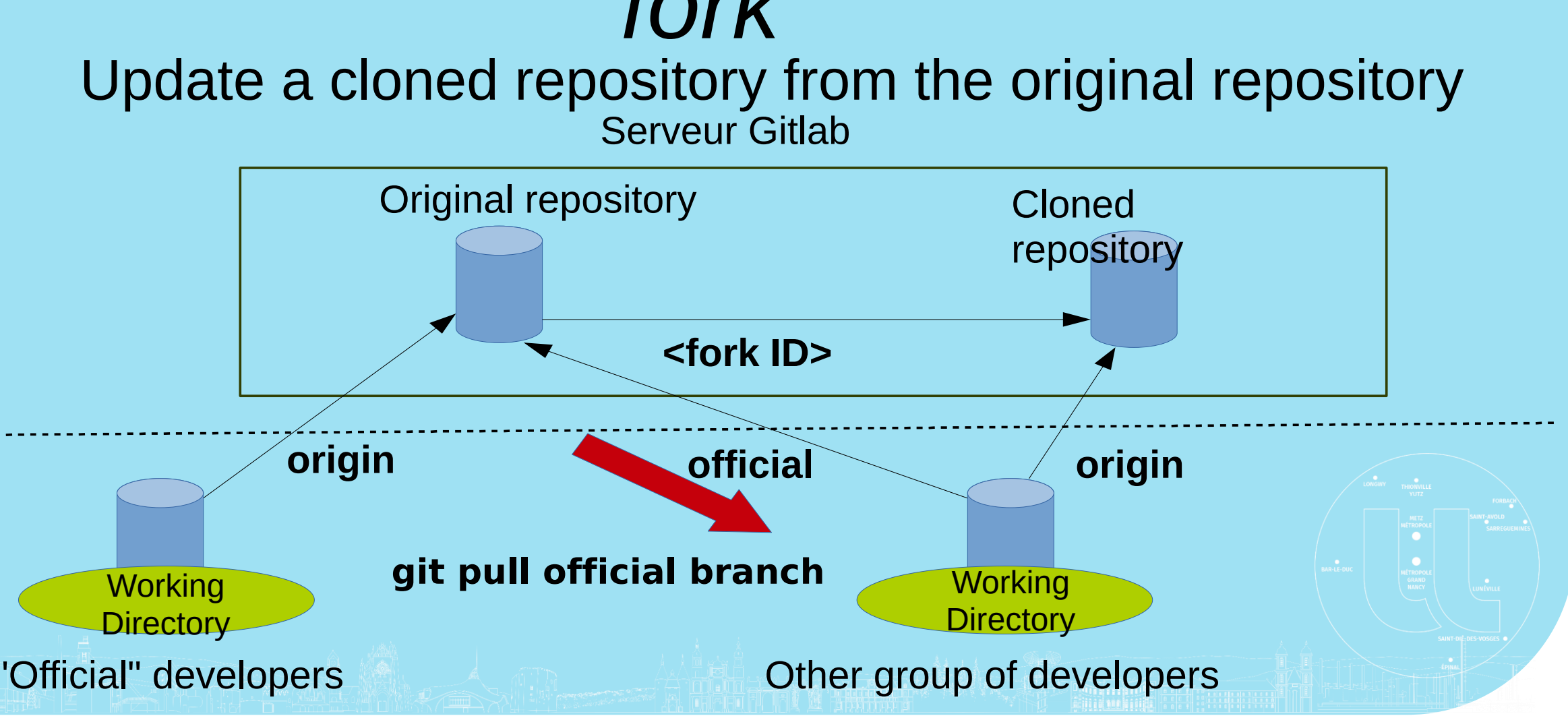
fork

Outside developers work in isolation



fork

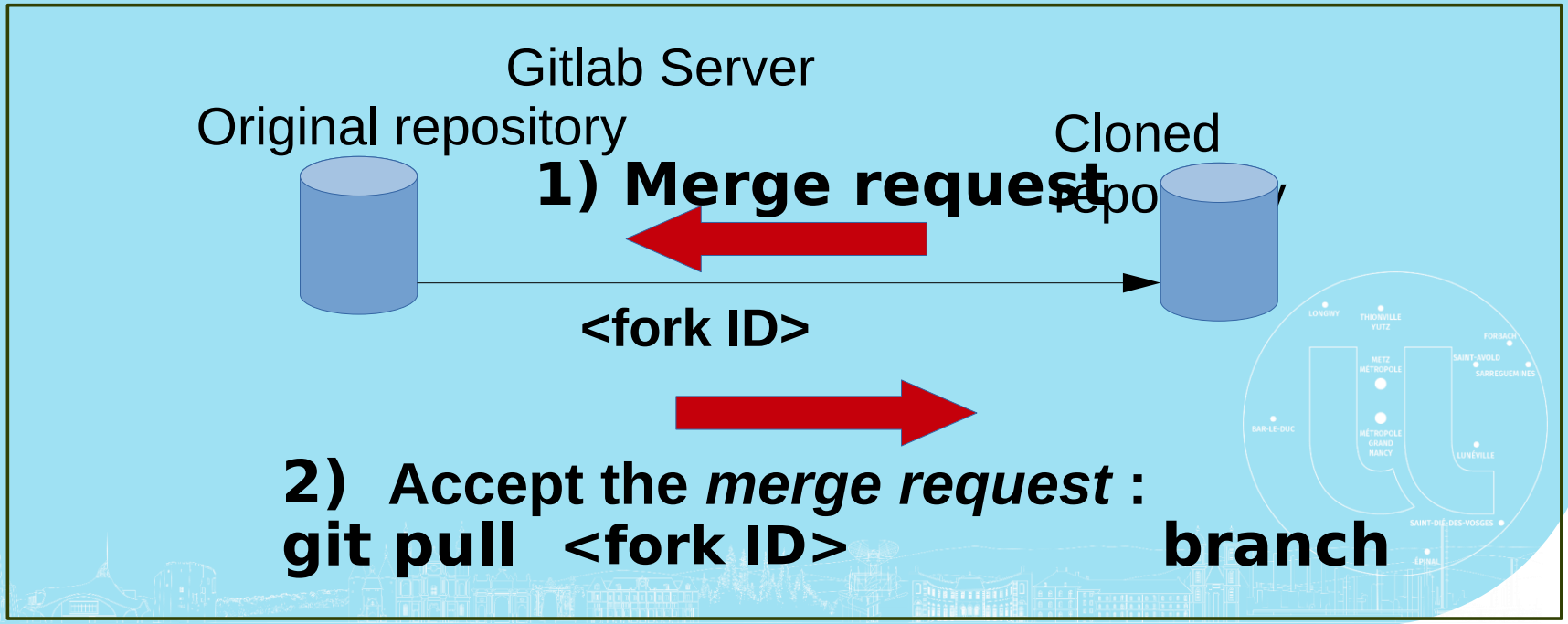
Update a cloned repository from the original repository



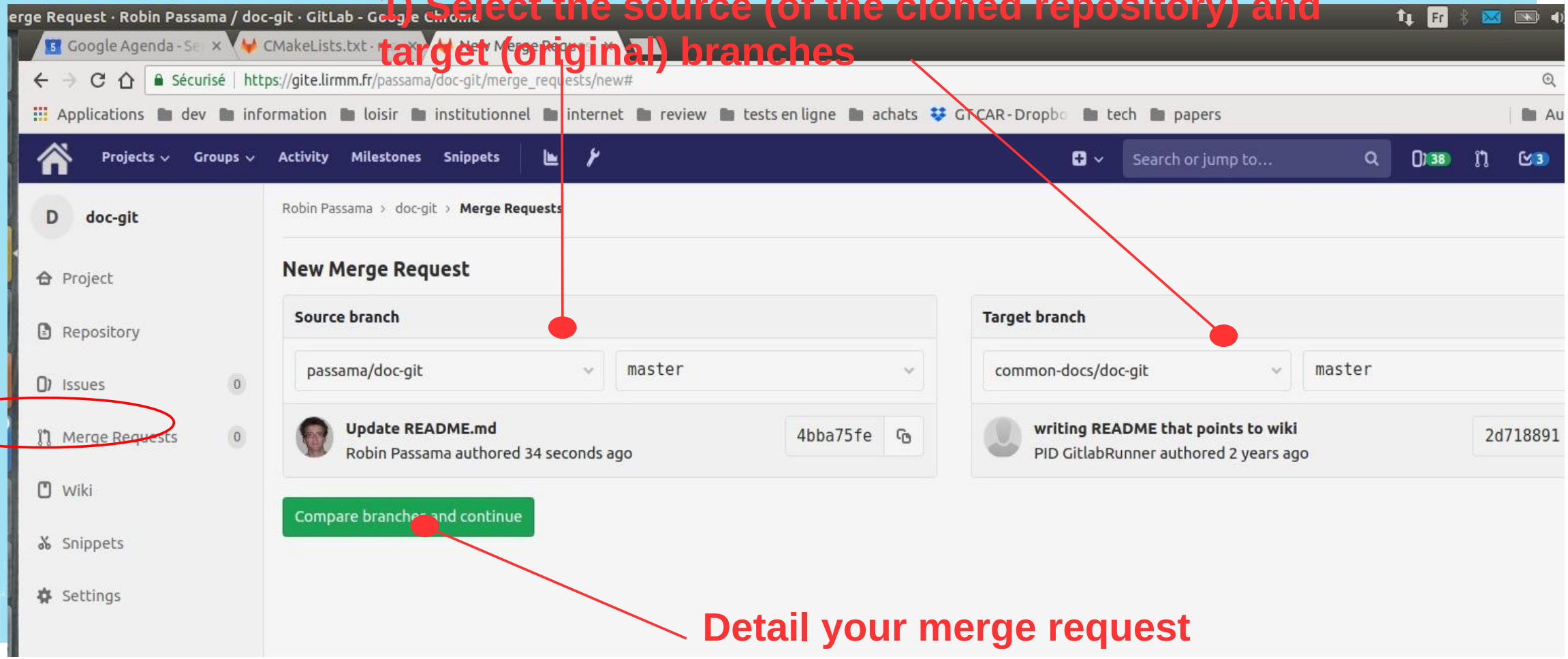
merge request

Proposal to **merge branches** from cloned repository to original repository

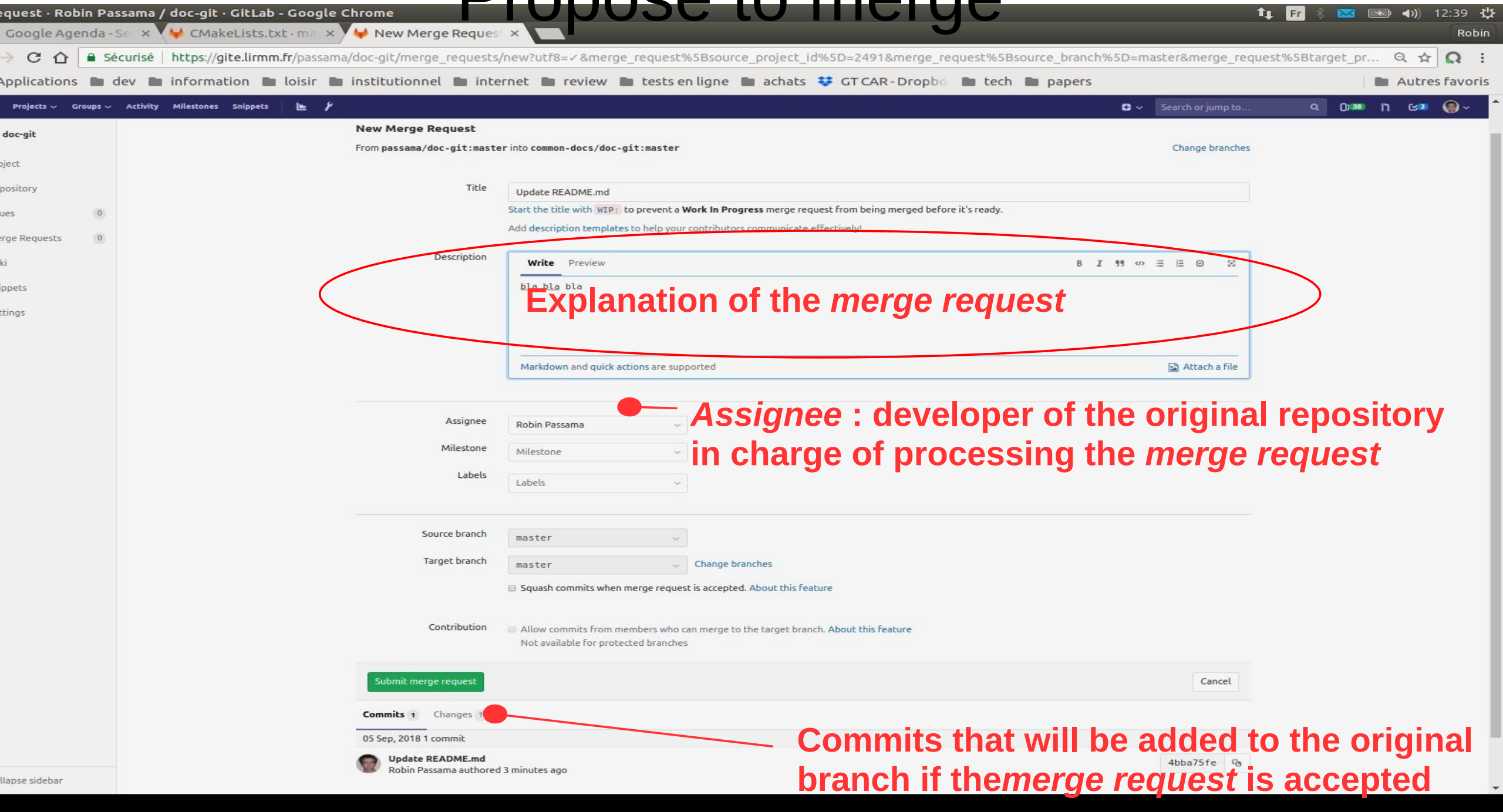
- Developers of the cloned repository propose the merging of a branch.
- Developers of the original repository decide whether the merge is completed.



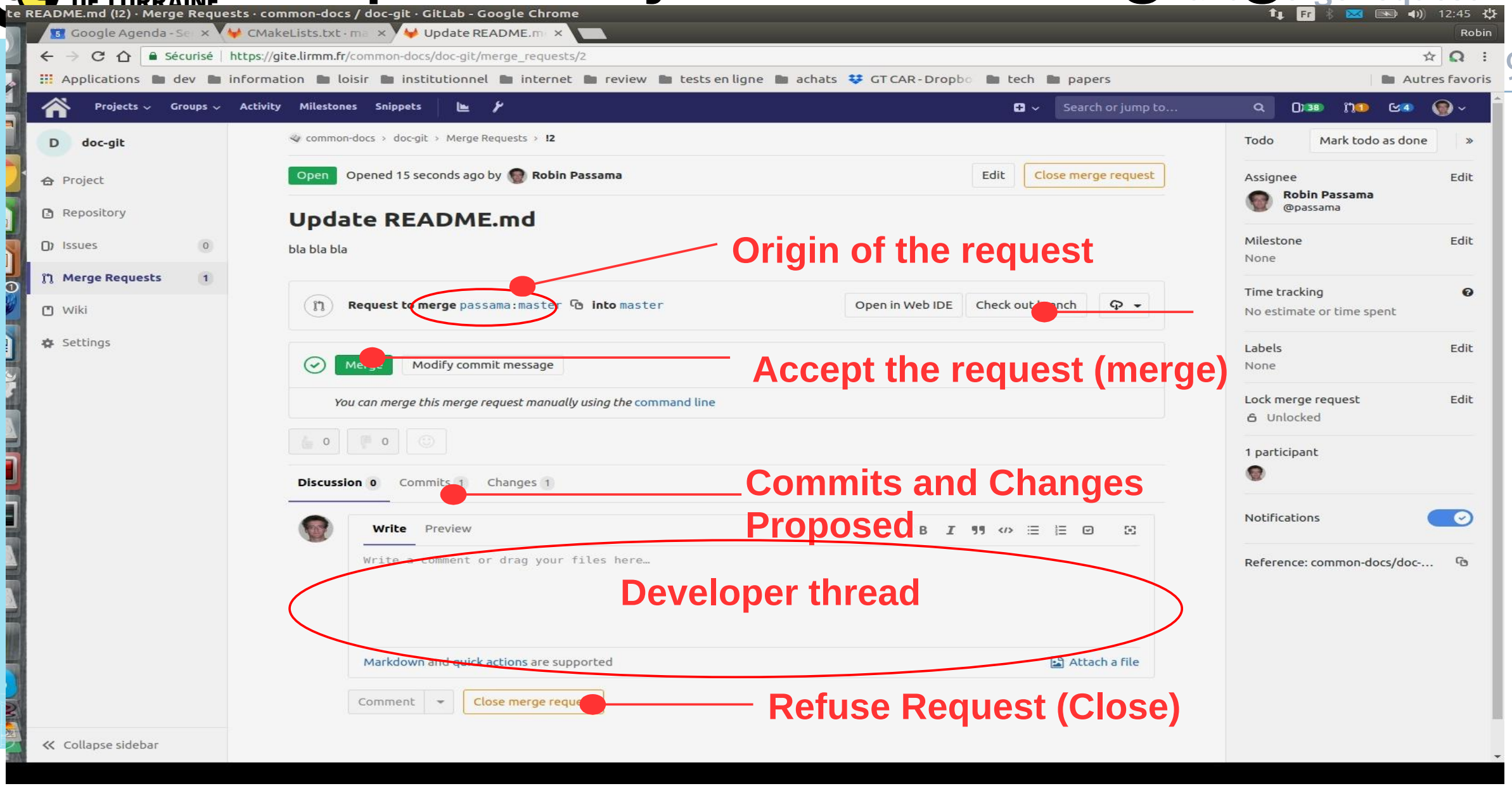
Propose to merge



Propose to merge



Accept or reject the merging



- The basic concepts and commands were shown...
- ... but there are a lot of refinements to discover!!
- Gitlab allows you to manage git repositories hosted on a server ...
- ... But it allows you to do many other things: continuous integration and deployment; Static site generation. wiki editing; etc.


Learn Git & GitLab
Version control for better software, together
Track changes
Never lose your work
Collaborate
Work together efficiently
Ship better code
Build great projects
Today's goals
Clone a repository
Create a branch
Make a commit
Push to GitLab
Open a merge request
Review code
Merge!
git
GitLab
Work locally
$ git clone <url>
$ git checkout -b feature
$ git add .
$ git commit -m "Add feature"
$ git push origin feature
Work together
Push your code
Open a merge request
Review & discuss
Merge to main
Code. Review. Improve. Together.
GitLab
AwesomeProject
Project overview
Issues 3
Merge requests 2
CI/CD
Repositories
Wiki
Members
Merge request
feature/login → main Open
Great work!
Let's review and merge this.
2 3
Small commits. Big progress.
$ git status
$ git add .
$ git commit -m "Update"
git push
Better code. Together.
Version Control
Software Engineering
GitLab
- Collaborate
- Review
- Build
- Deliver