Git@Equancy

This Handbook Is All You Need

Welcome to the world of version control and collaboration with GitLab! This guide outlines best practices for data scientists/engineers/analysts in Equancy to effectively use GitLab.

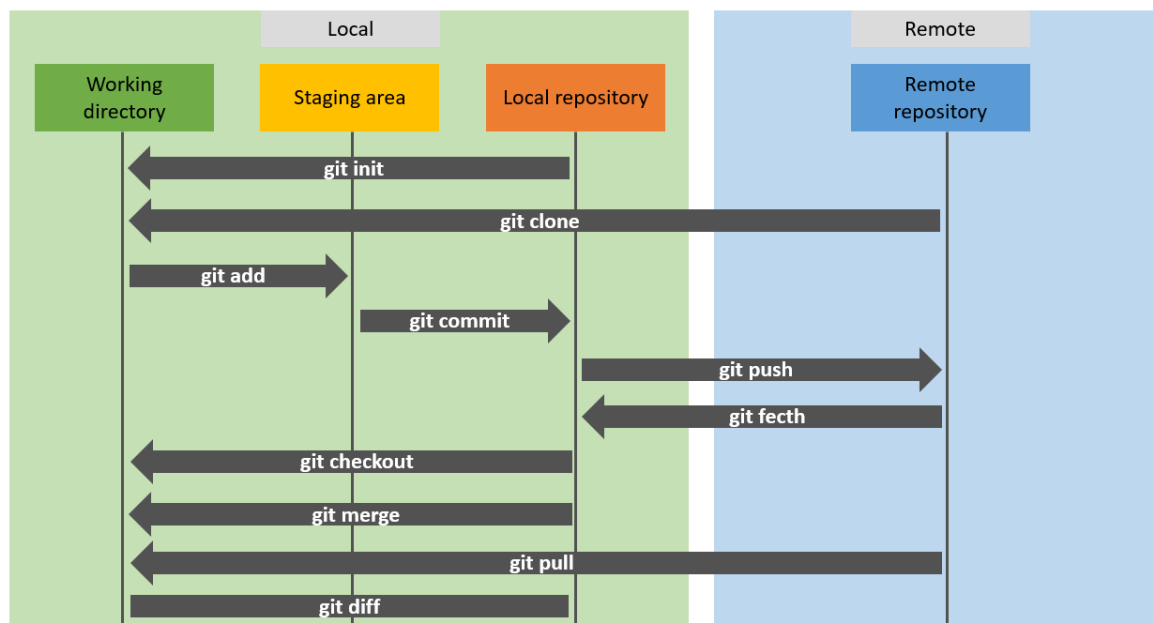You can open a cooler version of this document by following this link 😉

https://lime-desk-9f4.notion.site/Git-Equancy-335362d2afa04d4ca4e4ca8ff2177ad1

Table of contents

▼ 1. Introduction to GitLab

- Git is a distributed version control system that enables collaborative software development. It efficiently tracks code changes, allowing multiple developers to work simultaneously. With a decentralized structure, each contributor maintains a local repository with a comprehensive version history. It simplifies tracking modifications, managing project branches, and provides transparency about code changes and contributors.

- GitLab is a web-based platform for version control and collaboration based on Git. It allows you to manage your code, collaborate with team members, and automate your development workflow.

- General Git workflow diagram:

The diagram shows Git workflow between Local (Working directory, Staging area, Local repository) and Remote (Remote repository):
- git init
- git clone
- git add
- git commit
- git push
- git fecth
- git checkout
- git merge
- git pull
- git diff

- GitLab vs GitHub:

Aspect	GitLab	GitHub
Plan	GitLab provides integrated project planning, issue tracking, and a robust project management suite (Scrum, Kaban…).	GitHub also offers project management (Scrum) and issue tracking features but may require third-party integrations for more advanced project planning.
Create	GitLab offers a powerfull source code management suite (code review, wiki, WebIDE, search functionality and remote development).	GitHub also offers a similair source code management suite with additional forum/discussion acces to developers.
Verify	GitLab's CI/CD capabilities are integrated, supporting pipelines, secrets management, testing, and code quality checks.	GitHub also offers CI/CD, which allows for custom workflows and automation, supporting pipelines, and secrets management. Advanced CI/CD capabilities may require integrations.
Package	GitLab has a built-in container registry and supports proxy and firewall dependencies. It supports also Git LFS (Large File Storage).	GitHub offers GitHub Packages for container and package registries. And it supports also Git LFS.
Secure	GitLab includes built-in security scanning tools for code vulnerabilities and qualities and container scanning through CI/CD. And it ensure security on sensitive data with secret detention.	GitHub provides security features like GitHub Security Alerts but may rely on third-party integrations for some security aspects. It also supports secret detention.
Release	GitLab offers robust release management and deployment tools, including feature flags.	GitHub provides release management features, but the depth of these features can vary based on the integration of third-party tools.
Configure	GitLab provides auto DevOps, Kubernetes management, Infrastructure as Code (IaC), ChatOps and cluster cost management.	GitHub offers ChatOps and Kubernetes management through GitHub Action.

Aspect	GitLab	GitHub
Monitor	GitLab includes monitoring tools which allow incident management, tracing, error tracking, continuous verification.	GitHub offers monitoring and observability through GitHub Insights and integrations with third-party monitoring tools.
Govern	GitLab includes access control, auditing, and compliance features, making it suitable for governance and compliance requirements.	GitHub provides access control and auditing features, with added support from third-party apps and services.

See the following <u>link</u> for more details.

- Git UI

  - Based on command line: **Git bash,** installed with git and offers simple user interface. It is the most complete, and fast if you know the command. Avaliable <u>here</u>.

    To use the Git GUI built-in tools, run: `git-gui` for commiting or `gitk` for browsing.

  - Intergreted into a development environment: **Visual Studio Code, Jupyter Lab, Rstudio,** Google Colab (only with GitHub).

  - Dedicated to git: **GitKraken, GitHub Desktop, TortoiseGit**, see all GUI clients <u>here</u>.

▼ 2. Setting Up Your Workspace (for the first time)

- Install Git on your local machine if not already done.

- Configure your Git identity (username and email) with:

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

  You can list all the global configuration variables using `git config --global --list` to see your Git identity.

- To communicate with GitLab you have two possibilities:

  - HTTP protocol

  - SSH (Secure Shell) protocol

- **HTTPS protocol**

  HTTPS operates very similarly to the SSH protocols but runs over standard HTTPS ports, meaning it's often easier on the user than something like SSH, since you can use things like username/password authentication rather than having to set up SSH keys.

  For less sophisticated users, or users on systems where SSH is less common, this is a major advantage in usability. It is also a very fast and efficient protocol, similar to the SSH one.

  This is the basic protocol access, so you don't have to do anything more to use it!

- **SSH protocol**

  So why use SSH rather than HTTPS protocol? Because SSH is a more secure option, all the data transfered is encrypted and authenticated. Since SSH is more secure than entering credentials over HTTPS, it is recommended for business dealing with sensitive and critical data.

  It is also a very fast and efficient protocol, similar to the HTTPS one, but it can be a little tricky to set up.

Note that default SSH protocol use <u>RSA encryption</u>, but as mentioned in the <u>GitLab documentation on SSH key</u>, ED25519 encryption is more secure and performant than RSA.

- **Set up SSH keys**

  Before creating a SSH keys, you can check if a key pair already exists.

  1. Go to your home directory.

  2. Try to go to the `.ssh/` subdirectory, if it doesn't exist you are either not in the home directory, or you haven't used SSH before. In the latter case, you need to generate an SSH key pair.

  To create a SSH key pair, open a terminal and run the following command:

  ```
  ssh-keygen # default to rsa (key size depend on your system)
  ```

  Note that it will create a SSH using the default RSA encryption. You can use another encryption algorithm by using the `-t` flag.

  ```
  ssh-keygen -t ed25519 # generate key pair using ED25519 encryption algorithm
  ssh-keygen -t rsa -b 2048 # specify key size of 2048 bits (recommended)
  ```

  Then press `ENTER`. An ouput similar to this one is displayed:

  ```
  Generating public/private rsa key pair.
  Enter file in which to save the key (/c/Users/username/.ssh/id_rsa):
  ```

  You can accept the suggested filename and directory by pressing `ENTER`.

  Then, you will be asked to specify a passphrase (password):

  ```
  Enter passphrase (empty for no passphrase):
  Enter same passphrase again:
  ```

  Simply press `ENTER` twice for no passphrase, note that if you specify a passphrase, you will be required to enter it everytime you communicate with a remote repository (push, pull, etc.).

  Now that you have your SSH key pair, you need to add the SSH public key to your GitLab account. This is how to do it:

  1. Copy the content of your public key file. To do it, go to the `.ssh/` subdirectory and run:

     ```
     cat id_rsa.pub | clip # clip command is used to copy the content of the key
     ```

  2. Sign in to GitLab

  3. On the right sidebar, select your avatar and click on **Edit your profile**.

  4. On the left sidebar, select **SSH keys**.

  5. In the **Key** box, paste the contents of your public key. If you manually copied the key, make sure you copy the entire key, which starts with "ssh-rsa…" for example.

  6. In the **Title** box, type a description, like "Work laptop".

  7. You can **optionally** add an expiration date.

8. Select **Add key.**

To verify that you can connect, open a terminal and run this command:

```
ssh -T git@git.equancy.cloud
```

If this is the first time you connect, you should approve this connection, type `yes` and press `ENTER` . Then, you should receive a `Welcome to GitLab, @username!` message. That's it!

▼ 3. Creating a New Repository

- **Purpose**: Initialize a new project repository for your work.

- **When to do it**: Whenever you start a new project or need a new repository for code storage.

- **How to do it:**

  - If you don't have existing code: use GitLab Web UI to create a new project repository then clone it in your local workspace.

  - If you want to create a repo from a local folder: first create it in the web UI then go to your directory and run this in git bash:

    ```
    git init # -b <main> to set the default branch name to "main"
    git add . # add all existing files, you can customize this command
    git commit -m "Chore: Push existing project to GitLab"
    git remote add origin <repository_url> # connect your dir to GitLab
    # for SSH protocol, the repository url starts with git@...
    git push origin main
    ```

    Example of repository url with HTTPS protocol: *https://git.equancy.cloud/username/reponame.git*

    Example of repository url with SSH protocol: *git@git.equancy.cloud:username/reponame.git*

▼ 4. Cloning a Repository

- **Purpose**: Copy an existing repository from GitLab to your local machine for development.

- **When to do it**: Whenever you need to work on an existing project.

- **How to do it:**

  - Copy the SSH or HTTPS url from webpage of the repo.

  - Use the `git clone` command:

    ```
    git clone <repository_url>
    ```

▼ 5. Adding and Commiting Changes

- **Purpose**: Record and save changes made to your code with descriptive messages.

- **When to do it**: After making significant changes to your code or completing a specific task. Before committing your changes, it's essential to stage the modifications you want to include in the commit by adding them.

- **How to do it:**

  - Add: selectively choose which files or changes to include.

```
git add file1.js file2.py
```

- **Best practices:**
    - avoid using add *
    - add files related to one category: either bugfix or feature
    - if you want to unstage an added file, you can run:

    ```
    git restore --staged <file>
    ```

- Commit:

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

- The commit message should be structured as follows:
    - type: *fix*, *feat* or *chore* (documentation, refactoring, style etc). Type must be followed by *!* if it's a breaking change in the code

    ```
    <type>[optional scope]: <description>
    [optional body]
    [optional footer(s)]
    ```

    - *example:*

    ```
    feature: added outliers handling methods

    Select outliers detection and processing method from conf.yaml. Detect using 3-sigma rule. Process u
    sing pandas.interpolate

    Reviewed-by: Z
    ```

▼ 6. Branching and Merging

- **Purpose**: Branching allows teams of developers to easily collaborate inside of one central code base. When a developer creates a branch, the version control system creates a copy of the code base at that point in time. Changes to the branch don't affect other developers on the team.

- **When to do it**: Create a new branch when starting a new task, and merge when the task is ready for integration.

- **How to do it:**

Create a new branch

- Create a new branch for each feature or bug fix following this convention: ***<category/reference>***
    - **Category**: `feature` , `bugfix` , `hotfix` , or `dev` . **N.B.** if you're working alone on the repo, a dev branch is enough additionnally to the branch main. However, when working in a team, it's important that each developer creates their own branch and requests a merge to the repo admin as soon as the branch is ready. In the end of the project, all finished branches need to be merged into the main. Merged branches can then be removed.

- `feature` is for adding, refactoring or removing a feature

- `bugfix` is for fixing a bug

- `hotfix` is for changing code with a temporary solution (usually because of an emergency)

- `dev` is for experimenting outside of an issue/ticket

- **Reference**: reference of the issue/ticket you are working on

```
git checkout -b <branch_name> # <branch_name> = <category/reference>
```

Switch between two branches

- If you're in branch <current> and you want to switch to branch <branch> then use:

```
git checkout <branch>
```

Merge branches

- Merge branches into the another branch via merge requests in the web UI or via the command:

```
git merge <branch> # this applies commits in <branch> to current branch
```

- ***Example***: merging branch `dev` into branch `main` (assumes `dev` is already created)

```
git checkout main # switch to main
git pull # make sure main is updated with the latest commits
git merge dev # merge dev into main
```

▼ 7. Pulling and Pushing Changes

- **Purpose**: Share your local changes with the GitLab repository.

- **When to do it**: After committing your changes, push them to the repository to make them accessible to your team.

- **How to do it:**

  - Before pushing your commits, it's important to keep your local copy up-to-date with changes made by your team members. To fetch and incorporate the latest changes from the remote repository into your local copy, use:

```
git pull
```

  - To push your local changes to GitLab use:

```
git push origin <branch_name>
```

▼ 8. Handling Conflicts

- **Purpose**: Resolve conflicts that occur when merging code changes.

- **When to do it**: When GitLab indicates that there are conflicts during a merge, resolve them before completing the merge.

- **How to do it:**

  - **Pull the latest changes**

  - I**dentify conflict markers:** Open the file with conflicts in your code editor. You'll see conflict markers, such as `<<<<<<<`, `=======`, and `>>>>>>>`, which indicate the conflicting sections of the code. Manually resolve these conflicts by editing the file to keep the desired changes.

  - **Commit the resolution:** save the file and stage it using `git add`. Then, commit the resolution with a meaningful commit message using `git commit`.

  - **Complete the merge**: `git merge --continue`. This command finalizes the merge and records the resolution of the conflict.

  - **Push the changes**

▼ 9. Best Practices to Collaborate with Git

- **Purpose**: Handle working in collaboration with other people on the same project using merge request.

- **When to do it**: Everytime you work with someone else and you want to merge modification with others' work.

- **How to do it:**

  - Respect all the best practices listed above. Create a branch to develop a feature or fix a bug. Make the modification, add, commit and push to the newly branch.

  - Go on GitLab, and create a merge request to merge your branch with another branch (the main branch for example). Assign it to someone for a code review and eventualy to handle conflitcs.

    1. To create a merge request after the push, you can either click on the pop-up displayed by GitLab, or go on the left sidebar, click on **Merge request** (you may use the shortcut `g` + `m`) and click on **New merge resquest**.

    2. Then you can select a source branch and a target branch for the merge request. And click on **Compare branches and continue**.

    3. Add a title and a description to help contributors to understand effectively the merge request.

    4. Assign it to another contributor (you can assign it to yourself) using the field under **Assignee**. The contributor will be responsible for the first review of the merge request.

    5. You can add a **Reviewer** if you want a peer review before merging.

    6. Click on **Create merge request.** Note that by default, it will delete the source branch when merge request is accepted.

  - You can create a **Merge request** using Git `--push-option` or `-o` tag when you push your modification:

    ```
    git push -o <push_option>
    ```

    Or using the long format:

```
git push --push-option=<push_option>
```

You have a wide choice of push options, such as:

- `merge_request.create`

- `merge_request.target=<branch_name>`

- `merge_request.title="<title>"`

- `merge_request.description="<description>"`

- `merge_request.assign="<user_name>"`

- `merge_request.remove_source_branch`

To combine push options to accomplish multiple tasks at once, add tags one after the others. For example:

```
git push origin <branch_name> -o merge_request.create
-o merge_request.target=<branch_name> -o merge_request.title="<title>"
-o merge_request.description="<description>"
-o merge_request.assign="<user_name>" -o merge_request.remove_source_branch
```

- You can follow this <u>tutorial</u> to create a merge request or this <u>overall tutorial</u> about merge requests.

- You can view the merge requests assigned to you on the uppersidebar or using the shortcut `SHIFT` + `m` .
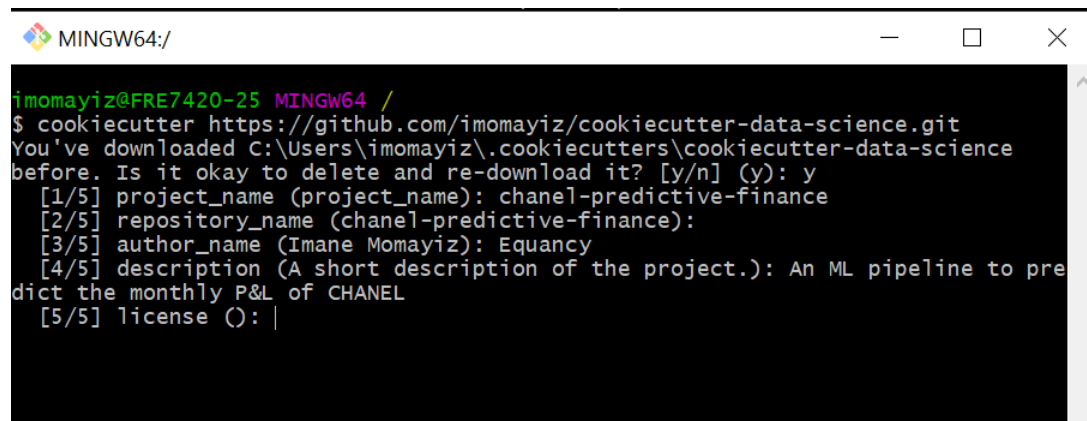
▼ 10. Collaboration Workflow @Equancy
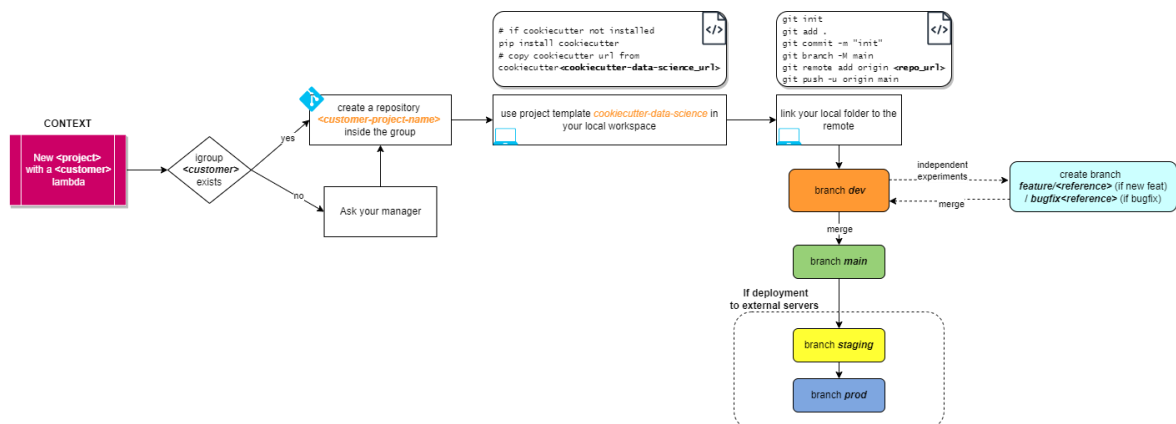
- **Practice example**: suppose you're working on a new data science project for CHANEL on predictive finance

  - *(gitlab UI)* navigate to the group CHANEL on gitlab

  - *(gitlab UI)* create a new repository named *chanel-predictive-finance*

  - *(cmd)* run:

```
pip install cookiecutter # if necessary
cookiecutter https://git.equancy.cloud/equancy/data-intelligence/cookiecutter-data-science-project.git
```

You will be asked to setup your project's name and other fields. If default values in between brackets suit you, then you can click on Enter. Otherwise, enter your value.

- *(cmd)* run:

```
git init
git add .
git commit -m "init"
git branch -M main
git remote add origin https://git.equancy.cloud/chanel/chanel-predictive-finance
git push -u origin main
```

- *(cmd)* create a branch dev (cf. section **branching**)

```
git checkout -b dev
```

- after each change in the branch *dev* (after pushing your commits), merge into the *main* branch (cf. section **merging**)

- if you want to make new experiments independently of your *dev* branch, you can create a new branch *feature/<reference>*, reference being the name of the feature. For eg. *feature/outliers-handling*

- if a new developer starts working on the repository, they should create a branch off the branch *main.* Please respect the naming convention as described in section **branching.** For eg:

```
git checkout -b feature/dash-app main
```

- if necessary, create your own branch off the dev branch and continue your developments. For eg. *bugfix/finetuning-pipeline*

- merge *feature/dash-app* and *bugfix/finetuning-pipeline* into *dev*

- merge *dev* into *main* and remove feature and bugfix branches

- keep collaborating by commiting and pushing your changes frequently, merging into dev (if necessary), then merging into main

- if you need to deploy your scripts to an external server, then create branches *staging* and *prod*. *Staging* is the intermediary branch between *main* and *prod.*

- **General guidelines**

- Follow our team's agreed-upon workflow.

- One commit = One change, i.e. either a bugfix, feature or chore. Avoid commiting multiple changes all at once.

- Keep the *main* branch stable.

- Avoid force-pushing to shared branches.

▼ 11. Common Git Pitfalls

1. **Commit all your changes in one time:** Using `git add .` to add all your changes in the staging area and then `git commit -m "your message"` will result in a single commit of all the changes, making it difficult to track your work. Commit often and make sure each commit represents a logical unit of work.

2. **Work on the main branch:** The main branch must to be stable, it is higly recommended to not working on the main/master branch. Always create a new branch for new work.

3. **Commit unnecessary/undesired files:** Sensitive data, large files, environment folders, bytes-compiled files, temporary files, etc. shoud not be commited to the repository. Make sure to use a proper .gitignore file.

4. **Commit undocumented:** Using meaningless commit messages can make it difficult for yourself and your team to understand the purpose of a change. Always follow the commit best pratices (see section 5. Adding and Committing Changes).

5. **Force flag:** Using the force flag `--force` can lead to undesired results such as overwrite other's work and thus data loss. Instead of using it, ask a more experienced colleague to help you solve your problem.

6. **Ignore merge conflicts:** Ignoring merge conflicts can lead to more conflicts. It's essential to address merge conflicts promptly to prevent code conflicts from accumulating.

7. **Copy and paste code:** Copying code changes outside of Git and pasting them into your working copy can result in missing changes in your commit history. Always use Git commands for branching and merging (it is designed for this).

8. **Ignore Git Hooks**: Git hooks provide automation capabilities but are often overlooked. Use them can help automate tasks like code linting, testing, quality, and more.

▼ 12. Tips

- **Stashing**:

  Stashing takes the dirty state of your working directory, that is, your modified tracked files and staged changes, and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch). It is helpful to prevent commit of half-done work when you want to switch branches for a bit to work on something else.

  ```
  git stash # saved working directory and index state
  git status # check the cleaned working directory
  ```

  It will display: *nothing to commit, working directory clean*

  To see which stashes you have stored using:

  ```
  git stash list
  ```

It will display something like that:

stash@{0}: WIP on main: 049d078 Create index file
stash@{1}: WIP on main: 21d80a5 Add number to log

You can reapply the last stash using `git stash apply` or select one by naming it `git stash apply stash@{1}`. Note that if you had a file in the staging area, it will not be restaged. To restaged it you can use `git stash apply --index`.

The apply option only tries to apply the stashed work — you continue to have it on your stack. To remove it, you can run git stash drop with the name of the stash to remove:

```
git stash drop stash@{0}
```

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

- **Alias:**

Git doesn't automatically infer your command if you type it in partially. If you don't want to type the entire text of each of the Git commands, you can easily set up an alias for each command using `git config`.

Here an example you may want to set up:

```
git config --global alias.unstage "reset HEAD~1"
```

However, maybe you want to run an external command, rather than a Git subcommand. In that case, you start the command with a `!` character. This is useful if you write your own tools that work with a Git repository. We can demonstrate by aliasing `git visual` to run `gitk` :

```
git config --global alias.visual "!gitk"
```

- **Tag:**

Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points (v1.0.0, v2.0.0 and so on), which is lighter than the git sha.

To create a tag you can use the following command:

```
git tag -a <tagname> # -m "commit associated to the tag"
```

Since tag are generally use to makr release points, an example is: v0.1.0 following the semantic versioning convention MAJOR.MINOR.PATCH.

You can list all the tags using `git tag` command, you can also list tags matching a specific pattern using `git tag -l "v.1.5.*"` it will match all the patch of the version 1.5. You can also dislay a specific tag using `git show <tag>`.

By default tags are not shared and stay on the local repository. You need to push them to the remote repository.

```
git push origin --tags
```

If you want to push a specific tag, you can also use the tag name: `git push origin <tagname>` .

Finally, you can delete a tag in the local repository using the following command `git tag -d <tagname>` and then delete it from the remote repository using `git push origin --delete <tagname>` .

▼ 13. Resources and Further Learning

- Equancy's cookiecutter template for DS projects: https://git.equancy.cloud/equancy/data-intelligence/cookiecutter-data-science-project.git

- Explore GitLab's official documentation and online tutorials https://docs.gitlab.com/.

- Explore previous internal dataschools and trainings from the folder *Toolkit_Cross_Sharing_Equancy/5_Data/Gitlab*

If you have any questions or need assistance, don't hesitate to reach out to your managers or our GitLab experts.