This documentation describes guidelines for the development workflow of a bioinformatics pipeline from software development to deployment in production. The guidelines capitalizes on the gitflow model. We assume that the reader is familiar with git and GitLab. However, we provide a brief Git tutorial and GitLab tutorial.

The documentation consists of three main sections:

1. The section Development workflow first describes the general principles of the development workflow we implemented. The branching model is presented along with the different use cases.

2. The section Nominal procedure then describes the first use case to develop a new release of the bioinformatics pipeline:
   - The General overview introduces the different steps of the use case,
   - The Technical procedure details all the actions and command lines step-by-step,
   - The Graphical synopsis of the procedure summarizes the technical actions.

3. The section Hotfix procedure then describes the second use case to correct a critical bug that has occurred in the production environment:
   - The General overview introduces the different steps of the use case,
   - The Technical procedure details all the actions and command lines step-by-step,
   - The Graphical synopsis of the procedure summarizes the technical actions.

### 1.1 Prerequisite

All the protocols require that you have set up a GitLab repository as explained in the section Create a new project in Gitlab.
1.2 Useful resources

Useful resources for biogitflow are available here:

- biogitflow documentation
- biogitflow template
- source of the biogitflow documentation

1.3 Cite us

This section describes the general principles of the development workflow we implemented. The branching model is presented along with the different use cases.

2.1 Description of the different steps

2.1.1 Step 1 - software development

This step includes several tasks:

- **write the code** to implement the expected functionalities from the specifications that have been formalized.
- **perform a set of testing** that consists of several levels:
  - **unit testing** confirms that a piece of code provides the expected output according to the input parameters. The developer is in charge of this testing.
  - **integration testing** checks that the interfaces of the different bioinformatics pipeline components are consistent with each other. It ensures that their integration allows the expected functionalities to be performed.
  - **system (or functional) testing** validates that the full bioinformatics pipeline works and fits well the user’s needs as they were expressed. A person other than the developer is ideally responsible for carrying out these tests (if the team size allows it).
  - **regression testing** checks that the correction of bugs or the development of new functionalities did not introduced defects in unchanged areas of the bioinformatics pipeline. New test cases are added whenever a new release is developed. The set of testing is entirely performed for each new release.

**Note:** For more details about software testing visit the International Software Testing Qualifications Board.
2.1.2 Step 2 - acceptance testing

Once all the testing from the Step 1 - software development is successful, the end-user must validate all the functionalities that have been developed in a mirror environment that is used in production.

Different representative use cases must be evaluated on real datasets. For a bioinformatics pipeline that includes several components (as it is generally the case for bioinformatics pipelines), it is necessary to check that no data are lost between the different processing steps. For example:

- if the bioinformatics pipeline annotates a list of genomic variants, then the number of variants used as input must be the same in the output and their genomic coordinates must remain unchanged after the processing.
- if the bioinformatics pipeline is a visualization interface, the integrity of the data that are displayed must be preserved.

If the end-user does not validate the developments, the Step 1 - software development starts again.

After the final validation by the end-user, an operational testing is set up using the tool jenkins.

**Important**: The operational testing is essential as it checks that the bioinformatics pipeline provides the expected results on a reference dataset (golden dataset) in the production environment. This testing is performed periodically and is typically run every time that the bioinformatics pipeline is used in the production environment. This ensures the reproducibility of the results.

2.1.3 Step 3 - check the installation process and new testing

This steps checks that the bioinformatics pipeline can be installed in a similar environment that is used in production.

Another set of testing is performed such that bugs can be corrected before installing the bioinformatics pipeline in production.

This step should be realized within a very short period of time. The bioinformatics pipeline is generally deployed in the production environment right after this step.

2.1.4 Step 4 - production deployment

During this last step, the new release of the bioinformatics pipeline with the new functionalities is installed in the production environment.

2.2 Multiple deployment environments

It is mandatory that the different steps that have been previously described are performed in separated environments for the following reasons:

- ensure that a stable version can be used in production,
- allow the end-users to validate a new release without any impact on both the version used in production or the version under development,
- allow the software developers to add new functionalities and modify the code without any impact on the end-users who are validating a new version and/or using the version currently in production.

Therefore, we use 3 deployment environments for the bioinformatics pipeline:

- **dev**: development environment
• **valid**: validation environment (also called pre-production)
• **prod**: production environment

Each bioinformatics pipeline has 3 environments that are accessible in dedicated folders. For example, the environments of the bioinformatics pipeline `foobar` would be located here:

- `/bioinfo/pipelines/foobar/dev`
- `/bioinfo/pipelines/foobar/valid`
- `/bioinfo/pipelines/foobar/prod`

**Note:** The deployment environments are not limited to the 3 environments that have been previously described. Indeed, each developer can deploy the bioinformatics pipeline in a **local** workspace to test the new functionalities. The deployment in the **dev** environment generally takes place when a preliminary set of testing has been successful.

### 2.3 Version control and branching model

The management of the different bioinformatics pipeline versions is based on different **git branches**. Each branch is used depending on the context and the step in the development workflow. The model we use is based on a **remote repository** that contains 4 branches:

- **devel**: contains the code of the current version under development. Note that the version under development may have not been yet deployed in the **dev** environment. The code remains on that branch while the bioinformatics pipeline has not successfully passed the Step 2 - acceptance testing.

- **release**: contains the code with both candidate and official releases. The **release** branch comes from the **devel** branch.

- **hotfix**: this branch is a mirror of the **release** branch and is used to patch the code that is in production. If a critical bug occurs in production, this branch is used to fix the issue.

- **master**: this branch is not used for development, it is only used to archive the code from the **release** and **hotfix** branches.

Among these four branches, the **release**, **master** and **hotfix** are protected branches. This means that only the developer with the **Maintainer** role in the GitLab repository can push code on the remote repository. Other developers have to use Create a Merge Request in Gitlab to submit their modifications to the **Maintainer**.

The developer will have to create local branches in the **local workspace** used for development whenever a new feature is implemented, a hotfix is resolved or problem occurred during the Step 3 - check the installation process and new testing. Therefore, the **local workspace** will contain:

- the branches from the **remote repository**.
- the local branches created by the developer to implement the new feature. These branches will not be sent onto the **remote repository**. They are named with the prefix **feature** and any meaningful suffix (e.g feature_star_mapper).
- the branches that developer will use to Create a Merge Request in Gitlab on protected branches. These branches will have either the prefix **release** or **hotfix** depending on the context.

The workflow across the different branches can be summarized in the graphic below:
Warning: No development must be initiated on a branch that exists on the remote repository. The developer always creates a local branch with the command `git checkout -b feature_1` (or any other more meaningful suffix). The modifications are committed on the local branch. Once validated, the modifications are merged on the branch from which the local branch was created.

### 2.4 Deployment and branching model

There is no bijection between the branches and the deployment environments as a version from a given branch can be deployed in different environments. However, only some combinations are allowed as described in the table below:

<table>
<thead>
<tr>
<th>branche</th>
<th>env:dev</th>
<th>env:valid</th>
<th>env:prod</th>
</tr>
</thead>
<tbody>
<tr>
<td>devel</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>release</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>hotfix</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>master</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
• ✓ is the normal case
• ❌ is a prohibited case
• ✔ is possible case
• ✓ is the normal case when both the Step 2 - acceptance testing and the Step 3 - check the installation process and new testing are successful
• ❗ is an exceptional use case

2.4.1 Steps of the workflow and deployment environment

During the different steps of the workflow, the deployment of the bioinformatics pipeline is performed in the following environments:

<table>
<thead>
<tr>
<th>Step 2 - software development</th>
<th>dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2 - acceptance testing</td>
<td>dev</td>
</tr>
<tr>
<td>Step 3 - check the installation process and new testing</td>
<td>valid</td>
</tr>
<tr>
<td>Step 4 - production deployment</td>
<td>prod</td>
</tr>
</tbody>
</table>

2.5 User roles and permissions

2.5.1 In the gitlab remote repository

In the remote repository on GitLab, the projects members will be assigned one of the following role:

- **Developer**: the user can push the developments on the non-protected branches. The letter D is used as an abbreviation for this role.
- **Maintainer**: in addition to the permissions with the Developer role, the user can push on the protected branches. The letter M is used as an abbreviation for this role.

2.5.2 In the deployment environments

For the deployment, two roles are considered:

- **UD**: the user can deploy in the dev environment.
- **UVP**: the user can deploy in the valid and prod environments.

**Note**: Whenever necessary, the roles that are required to perform the different actions will be mentioned. For example, M+UVP means that the user must have the Maintainer role in gitlab and can deploy in both the valid or prod environments.
CHAPTER 3

Nominal procedure

The section describes the first use case to develop a new release of the bioinformatics pipeline.

3.1 General overview

First we provide an overview of the development workflow in the nominal mode.

3.1.1 Step 1 - software development

- The developer (D) reads the specifications of the new feature to be implemented in the GitLab issue or writes them in a new GitLab issue using the appropriate template.
- The developer (D) creates a local branch named **feature** (it is recommended to give a meaningful name such as **feature_star_mapper**) on the local workspace from the **devel** branch to implement the new expected functionalities, test the code, commit the modifications, merge them and push the code on the remote repository on the **devel** branch.
- The developer (D) deploys the code from the **devel** branch either in a personal environment for testing or in the **dev** environment to perform unit, integration, system and regression testing.
- The developer (D) checks that the expected functionalities have been correctly implemented.
- If the testing is successful, it is possible to move to **Step 2 - acceptance testing**.

3.1.2 Step 2 - acceptance testing

- The user (D+UD) creates a new GitLab issue with the label **validation** using the appropriate template. This GitLab issue allows the tracking of all the discussions with the end-users who will validate the new release.
- The end-user can start the acceptance testing of the new release:
– either, the end-user validates the new release,
– or the end-user does not validate the new release. Thus, the developer (D) goes back to Step 1 - software development to implement the corrections until acceptance by the end-user.

- The user (M+UVP) implements an operational testing in jenkins. The test is named with the name of the remote repository and a suffix _DEV is added (e.g. foobar_DEV). The operational testing is launched.

3.1.3 Step 3 - check the installation process and new testing

- The user (M+UVP) brings the content of the devel branch into the release branch.

- The user (M+UVP) creates a GitLab issue with the label using the appropriate template in order to track the different steps of the installation process. The ID of the GitLab issue opened for the of the release is also reported for tracking.

- The user (M+UVP) deploys the code from the release branch in the valid environment.

- The user (M+UVP) checks that the installation is successful.

- The user (M+UVP) implements an operational testing in jenkins. The test is named with the name of the remote repository and a suffix _VALID is added (e.g. foobar_VALID). The operational testing is launched.

- Possibly, new corrections must be implemented on the release branch before the deployment in production. In this case:
  - The developer (D) creates a local branch names release-id_version-user from release branch. The id_version is the version number of the new release and the user is the unix login of the developer (e.g. release-version-1.2.3-phupe).
  - The developer (D) develops the corrections.
  - The developer (D) pushes the code from the local branch release-id_version-user on the remote repository.
  - The developer deploys the code from the release-id_version-user branch either in a personal environment for testing or in the dev environment to perform unit, integration, system and regression testing.
  - Once the code validated, the developer (D) has to Create a Merge Request in Gitlab from the release-id_version-user branch on the release branch. The Merge request is assigned to as user who has the Maintainer role.
  - The user (M+UVP) reviews and accepts the Merge Request.

- Once all the testing successful, the user (M+UVP) adds a tag on the release branch with the new version number.

3.1.4 Step 4 - production deployment

- Once validated by the end-user, the user (M+UVP) deploys the code in the prod environment from the release branch.

- The user (M+UVP) implements an operational testing in jenkins. The test is named with the name of the repository (e.g. foobar). The operational testing is started.

- Once the deployment is successful, the user (M+UVP) brings the content of the release branch into the master branch for archiving.
• The user (M+UVP) brings the content of the release branch into the hotfix branch.

• If some modifications were committed on the release branch, the user (M+UVP) brings the content of the release branch into devel branch such that the corrections can be integrated in the future release. Note that possible conflicts may exist on some pieces of the code. They will have to be resolved before merging thus requiring the help from the other developers involved in the modifications.

• If needed, the user (M+UVP) deploys the code from release branch in the dev environment, such that at this stage of the workflow, the same commit ID of the bioinformatics pipeline could be deployed in the dev, valid and prod environment.

• The user (M+UVP) closes the GitLab issue that were previously opened.

3.2 Technical procedure

3.2.1 Step 1 - software development

Code writing

• The developer (D) reads the specifications of the new feature to be implemented in the GitLab issue or writes them in a new GitLab issue using the template new_feature.

• The developer (D) clones the remote repository:

```
git clone git@gitlab.com:biogitflow/biogitflow-demo.git
```

• The developer (D) enters the folder in the local workspace:

```
cd my_project
```

• The developer (D) switches to the devel branch in the local workspace, gets the last modifications from the remote repository and checks that the right branch is used:

```
git checkout devel
git pull
git submodule update --init --recursive # if your git repo has submodules
git branch -vv
```

• The developer (D) creates a local branch named feature (it is recommended to give a meaningful name such as feature_star_mapper) and uses it to implement the new expected feature:

```
git checkout -b feature
git branch -vv
```

• The developer (D) implements the new feature.

• The developer (D) tests the code (unit testing).

• The developer (D) checks which files have been modified:

```
git status
```

• The developer (D) specifies which files to be added in the staging area:
The developer (D) commits the modifications with an explicit message using the *Naming convention for the commit messages*. The GitLab issue number is also reported in the commit message:

```
git commit -m "[MODIF] Sorting algorithm optimisation (Issue #11)"
```

Once the developments have been completed and committed on the local feature branch, the developer (D) switches to the devel branch:

```
git checkout devel
```

The developer (D) updates the local workspace with the remote repository:

```
git pull
git branch -vv
```

The developer (D) brings the content of the feature branch into the devel branch:

```
git merge feature
```

If needed, the developer (D) resolves the conflicts and checks that the code is still functional after the merge.

The developer (D) pushes the new version from the devel branch on the remote repository:

```
git push origin devel
```

The developer (D) deploys the code from the devel branch either in a personal environment for testing or in the dev environment to perform unit, integration, system and regression testing.

The developer (D) checks that the expected functionalities have been correctly implemented.

The developer (D) checks that all the modified files have been committed and pushed on the remote repository:

```
git status
```

The developer (D) deletes the local branch:

```
git branch -d feature
```

### Deployment in the dev environment

The user (D+UD) deploys the bioinformatics pipeline in the dev environment from the devel branch using the ad-hoc deployment scripts. *The deployment is only based on a commit ID.*

**Danger:** Tags with version number are not used to deploy the code but only a commit ID. Indeed, a tag can be easily removed or moved in git thus it is not a reliable information for tracking. This is why commit ID are used for deployment.

At this stage, the current version under development, is deployed in `/bioinfo/pipelines/foobar/dev` and the file `/bioinfo/pipelines/foobar/dev/version` contains the commit ID that has been deployed.
Testing

- The user (D+UD) performs unit, integration, system and regression testing. Other developers who implemented the new feature can contribute to validate all or part of the testing. If the testing is successful, we move to the Step 2 - acceptance testing, if not, we go back to the Step 1 - software development.

### 3.2.2 Step 2 - acceptance testing

#### Acceptance testing by end-users

- The user (D+UD) creates a GitLab issue using the template validation.
- The title of the GitLab issue must indicate the characteristics of the version to be validated.
- The description in the GitLab issue lists the new features/modifications that have to be communicated to the end-users.
- At the end of the description, a line such as fyi: @user1, @user2, @user3 is added such that all the persons involved in the validation process receive a notification.
- The GitLab issue is labeled with validation.
- The GitLab issue is assigned to a user (M+UVP).
- The user (D+UD) or the user (M+UVP) sends an email to all the persons who are involved in the validation process.
- The end-users can start the acceptance testing process:
  - either the end-users validate the new release,
  - or the end-users do not validate the new release. Then, the reason are tracked in the GitLab issue that has been created. We go back to Step 1 - software development. The developer (D) develops the modifications requested by the end-users on a local feature branch derived from the devel branch. The process is iterated until the validation by the end-users. The same GitLab issue is used to track all the information during the validation process until the final validation.

#### Implement and launch the operational testing on Jenkins

- The user (M+UD) modifies (if needed) the script to launch the operational testing (e.g. test/run-test.sh).
- If no operational testing exists, the user (M+UD) creates a new project in jenkins with the name of the repository and a suffix _DEV (e.g. foobar_DEV).
- The user (M+UD) modifies the parameters of the operational testing (for example, a new reference dataset may be used for this purpose, etc.).
- The user (M+UD) launches the operational testing.
- If the operational testing fails (the bioinformatics pipeline does not work or is not reproducible), go back to the Step 1 - software development.
Update the CHANGELOG

Note: The CHANGELOG file provides a simple history of the different versions of the bioinformatics pipeline. The version numbers are listed by decreasing order.

- A version number is added in the CHANGELOG using the following naming convention: version-x.y.z.
- Comments are added in the CHANGELOG to describe the most relevant functionalities added to the new release.

The CHANGELOG is divided into 3 sections:
- NEW FEATURES
- SIGNIFICANT USER-VISIBLE CHANGES
- BUG FIXES

Example of CHANGELOG file:

```
version-1.2.1

BUG FIXES
[BUGFIX] segfault corrected

NEW FEATURES
[MODIF] optimization of the sort function
[DOC] update of the end-user documentation

version-1.2.0

NEW FEATURES
[MODIF] use of GRCh38 reference genome
```

- The user (M+UD) updates the devel branch to get the last modifications from the remote repository and checks that the right branch is used:

```
git checkout devel
git pull
git branch -vv
```

- If needed, the user (M+UD) asks the other developers to define what comments should be added in the CHANGELOG and pushes the modifications on the remote repository:

```
git add CHANGELOG
git commit -m "[DOC] information about the version-1.2.3 added in the CHANGELOG"
git push origin devel
```

### 3.2.3 Step 3 - check the installation process and new testing

Bring the content of the devel branch into the release branch

- The user (M+UVP) switches to release branch from the local workspace and updates it:

```
git checkout release
git status
```

(continues on next page)
biogitflow Documentation, Release 1.0.1

(continued from previous page)

```
git pull
git branch -vv
```

- The user (M+UVP) brings the content of the `devel` branch into the `release` branch using the option `--no-ff` to avoid the fast-forward mode. This option will produce a new commit ID with a specific message to describe and track the merge:

```
git merge --no-ff devel
```

- The user (M+UVP) pushes the modifications from the `release` branch on the remote repository:

```
git push origin release
```

- The user (M+UVP) creates a GitLab issue using the template `deploy_in_prod`:
  
  - The GitLab issue is labeled with ![production](image).  
  
  - The number of the GitLab issue opened with the label ![validation](image) along with the numbers of the other Issues that describe the new features of the new release are added for tracking in this new GitLab issue.
  
  - The different steps of the deployment in production are recorded to track all the information about the installation along with the information about the data that have been used.
  
  - Every time a new step is performed for the installation, the new information are appended in the GitLab issue.

**Deployment in the valid environment**

The user (M+UVP) deploys the pipeline in the `valid` environment from the `release` branch using the ad-hoc deployment scripts. The deployment is only based on a commit ID.

**Danger:** Tags with version number are not used to deploy the code but only a commit ID. Indeed, a tag can be easily removed or moved in git thus it is not a reliable information for tracking. This is why commit ID are used for deployment.

At this stage, the current version under development, is deployed in `/bioinfo/pipelines/foobar/valid` and the file `/bioinfo/pipelines/foobar/valid/version` contains the commit ID that has been deployed.

**Test the version deployed in the valid environment**

The user (M+UVP) tests the bioinformatics pipeline.

**Implement and launch the operational testing in Jenkins**

- The user (M+UVP) modifies (if needed) the script to launch the operational testing (e.g. `test/run-test.sh`).

3.2. Technical procedure
biogitflow Documentation, Release 1.0.1

• If no operational testing exists, the user (M+UVP) creates a new project in jenkins with the name of the repository and a suffix _VALID (e.g. foobar.VALID).
• The user (M+UVP) modifies the parameters of the operational testing (for example, a new reference dataset may be used for this purpose, etc.).
• The user (M+UVP) launches the operational testing.
• If the operational testing fails (the bioinformatics pipeline does not work or is not reproducible), go back to the Development of corrections if needed.

Development of corrections if needed

In most of the cases, the deployment in the valid environment is very simple and quick. However, it might be necessary to correct some bugs before the deployment in production. In that case:

• The developer (D) checkouts and updates the release branch from the local workspace:

```bash
git checkout release
git status
git pull
git branch -vv
```

• The developer (D) creates a local branch named release-id_version-user (e.g. release-version-1.2.3-phupe), the release-version is retrieved from the CHANGELOG file) and uses it for the developments:

```bash
git checkout -b release-id_version-user # change the id_version and user values
```

• The developer (D) implements the corrections, tests the modifications and commits them (see Code writing for the details about the command lines).
• The developer (D) temporarily pushes the local branch on the remote repository:

```bash
git push origin release-id_version-user # change the id_version and user values
```

• The developer (D) deploys the code from the release-id_version-user branch either in a personal environment for testing or in the dev environment to perform unit, integration, system and regression testing.
• Once the code validated, the developer (D) creates a Create a Merge Request in Gitlab from the release-id_version-user branch on the release branch using the template merge_request_template.md. The Merge request is assigned to a user with the Maintainer role.
• The user (M+UVP) reviews and accepts the Merge Request.
• The user (M+UVP) updates the release branch from the local workspace:

```bash
git checkout release
git status
git pull
git branch -vv
```

• The user (M+UVP) go back to Deployment in the valid environment.

Add a tag with the version number

Once the new release has been validated and the installation in the valid environment is successful, the user (M+UVP) adds a tag (using the same version number that has been written in the CHANGELOG file) on the current HEAD:
tag-version4prod.sh -t version-1.2.3
git push --tags

Note: The script tag-version4prod.sh checks that the tag name is consistent with what was mentioned in the CHANGELOG and add the tag.

3.2.4 Step 4 - production deployment

Update the local repository with the version to deploy

It is likely that the local repository is not up-to-date anymore especially if a Merge Request has been submitted on GitLab. The user (M+UVP) updates the local workspace:

```
git checkout release
git status
git pull
```

Deployment in the prod environment

The user (M+UVP) deploys the bioinformatics pipeline in the prod environment from the release branch using the ad-hoc deployment scripts. The deployment is only based on a commit ID. The last commit ID from the release branch must be deployed.

Danger: Tags with version number are not used to deploy the code but only a commit ID. Indeed, a tag can be easily removed or moved in git thus it is not a reliable information for tracking. This is why commit ID are used for deployment.

At this stage, the current version under development, is deployed in /bioinfo/pipelines/foobar/prod and the file /bioinfo/pipelines/foobar/prod/version contains the commit ID that has been deployed.

Implement and launch the operational testing in Jenkins

- The user (M+UVP) modifies (if needed) the script to launch the operational testing (e.g. test/run-test.sh).
- If no operational testing exists, the user (M+UVP) creates a new project in jenkins with the name of the repository (e.g. foobar).
- The user (M+UVP) modifies the parameters of the operational testing (for example, a new reference dataset may be used for this purpose, etc.).
- The user (M+UVP) launches the operational testing.
- If the operational testing fails (the bioinformatics pipeline does not work or is not reproducible), restore the previous release and go back to the Development of corrections if needed.
**Bring the content of the release branch into the master branch**

- At this stage, there is a stable code on the `release` branch that has been tested, validated and successfully installed in the `prod` environment.

- The user (M+UVP) checkouts and updates the `master` branch:

```bash
git checkout master
git status # everything must be cleaned
git pull
git branch -vv
```

- The user (M+UVP) brings the content of the `release` branch into the `master` branch using the option `--no-ff` to avoid the fast-forward mode. This option will produce a new commit ID with a specific message to describe and track the merge:

```bash
git merge --no-ff release # can be a bit verbose
git status # must be cleaned
git branch -vv
```

- The `git status` must absolutely says something like this (otherwise, ask for help before moving forward):

```bash
# On branch master
# Your **branch is ahead of 'origin/master' by** 113 commits.
# (use "git push" to publish your local commits)
# nothing to commit, working directory clean
```

- The user (M+UVP) pushes the modifications on the remote repository:

```bash
git push origin master
```

**Bring the content of the release branch into the hotfix branch**

- At this stage, there is a stable code on the `release` branch that has been tested, validated and successfully installed in the `prod` environment and merged with the `master` branch.

- The user (M+UVP) checkouts and updates the `hotfix` branch:

```bash
git checkout hotfix
git status # must be cleaned otherwise, commit or stash your modifications
git pull
git branch -vv
```

- The user (M+UVP) brings the content of the `release` branch into the `hotfix` branch using the option `--ff-only` to only use the fast-forward mode in order to have the exact same commit ID between the `release` branch and the `hotfix` branch:

```bash
git merge --ff-only release # always use the options --ff-only
git status # must be clean
git branch -vv
```

- The `git status` must absolutely says something like this (otherwise, ask for help before moving forward):

```bash
# On branch master
# Your **branch is ahead of 'origin/master' by** 113 commits.
```

(continues on next page)
# (use "git push" to publish your local commits)
#
# nothing to commit, working directory clean

- The user (M+UVP) pushes the merge on the **remote repository**.

```
    git push origin hotfix
```

## Bring the content of the release branch into the devel branch

- At this stage, there is a stable code on the **release** branch that has been tested, validated and successfully installed in the **prod** environment and merged with the **master** and the **hotfix** branches.

- This step is necessary if some commits have been done on the **release** branch (this occurs only if there was a **Development of corrections if needed**).

- The user (M+UVP) checkouts and updates the **devel** branch:

```
    git checkout devel
    git status # must be cleaned otherwise, commit or stash your modifications
    git pull
    git branch -vv
```

- The user (M+UVP) brings the content of the **release** branch into the **devel** branch:

```
    git merge --no-ff release # may be verbose
    git status # may say something
    git branch -vv
```

- If the **devel** branch has been modified in the meantime, git will try to merge the modifications from the **release** branch.

- If some files cannot be merged automatically, they will appear to have **conflicts** in the output of the **git status**:

```
# On branch devel
# You have unmerged paths.
# (fix conflicts and run "git commit")...
# (use "git add ...") to mark resolution
# both modified:build.xml
```

- The conflicts have to be resolved manually. In that case, ask the help from the other developers.

- The files with resolved conflicts must be added to the staging area, committed, and the merge must be sent on the **remote repository**:

```
    git push origin devel
```

- The user (M+UVP) closes the GitLab issue and the GitLab issue that have been opened.

3.2. **Technical procedure**
Back on the devel branch

For security reason, the user (M+UVP) switches on the devel branch to avoid any risk of code modification on the master branch:

```bash
git checkout devel
git pull
git branch -vv
```

### 3.3 Graphical synopsis of the procedure
### Step 1 - Software Development

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Roles</th>
<th>Branches</th>
<th>Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="c" alt="C" /></td>
<td><img src="d" alt="D" /></td>
</tr>
</tbody>
</table>

If testing successful go to Step 2, if not go to Step 1.1

### Step 2 - Acceptance Testing

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Roles</th>
<th>Branches</th>
<th>Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="o" alt="O" /></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="p" alt="P" /></td>
<td><img src="**" alt="**" /></td>
</tr>
</tbody>
</table>

If end-users validate and testing successful, go to Step 3.1, if not go to Step 1.1

### Step 3 - Check the Installation Process and New Testing

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Roles</th>
<th>Branches</th>
<th>Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="m" alt="M" /></td>
<td><img src="p" alt="P" /></td>
</tr>
<tr>
<td>3.2</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="d" alt="D" /></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="o" alt="O" /></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td><img src="x" alt="X" /></td>
<td></td>
<td><img src="c" alt="C" /></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td><img src="x" alt="X" /></td>
<td></td>
<td><img src="p" alt="P" /></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td><img src="x" alt="X" /></td>
<td></td>
<td><img src="d" alt="D" /></td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="m" alt="M" /></td>
<td><img src="p" alt="P" /></td>
</tr>
</tbody>
</table>

If testing successful, go to Step 3.8, if not go to Step 3.4

### Step 4 - Production Deployment

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Roles</th>
<th>Branches</th>
<th>Environments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td><img src="x" alt="X" /></td>
<td><img src="x" alt="X" /></td>
<td><img src="d" alt="D" /></td>
<td></td>
</tr>
</tbody>
</table>
This section describes the second use case to correct a critical bug that has occurred in the production environment.

4.1 General overview

First we provide an overview of the development workflow when a critical bug occurs in the prod environment.

4.1.1 Step 1 - software development

- The developer (D) reads the bug report or writes it in a new GitLab issue using the appropriate template.
- The developer (D) creates a local branch named hotfix-id_commit-user on the local workspace from the hotfix branch. The id_commit is the commit number of the version that has been deployed in the prod environment that needs to be fixed, and user is the unix login of the developer (e.g. hotfix-aa12bb34-phupe).
- The developer develops the corrections.
- The developer (D) temporary pushes the code from the local branch hotfix-id_commit-user on the remote repository.
- The developer (D) deploys the code from hotfix-id_commit-user branch either in a personal environment for testing or in the dev environment to perform unit, integration, system and regression testing.
- The developer (D) checks that the bug has been fixed.
- If the testing is successful, it is possible to move to Step 2 - acceptance testing.

4.1.2 Step 2 - acceptance testing

- The user (D+UD) creates a new GitLab issue with the label using the appropriate template. This GitLab issue allows the tracking of all the discussions with the end-users who will validate the new release.
• The end-user can start the acceptance testing of the corrected release:
  – either the end-user validates the corrections,
  – or the end-user does not validate the corrections. Thus, the developer (D) goes back to Step 1 - software development to implement the corrections on the hotfix-id_commit-user branch until acceptance by the end-user.

• The developer (D) has to Create a Merge Request in Gitlab from the hotfix-id_commit-user branch on the hotfix branch. The Merge request is assigned to as user who has the Maintainer role.

• The user (M+UVP) reviews and accepts the Merge Request.

### 4.1.3 Step 3 - check the installation process and new testing

• The user (M+UVP) creates a GitLab issue with the label using the appropriate template in order to track the different steps of the installation process. The ID of the GitLab issue regarding the validation of the release is also reported for tracking.

• The user (M+UVP) deploys the code from the hotfix release branch in the valid environment.

• The user (M+UVP) checks that the installation is successful, launches a set of testing, in particular the operational testing.

• Once the validation successful, the user (M+UVP) adds a tag on hotfix branch with the new version number.

### 4.1.4 Step 4 - production deployment

• Once validated by the end-user, the user (M+UVP) deploys the code in the prod environment from the hotfix branch.

• The user (M+UVP) launches the operational testing dedicated to the prod environment (e.g. foobar).

• Once the deployment is successful, the user (M+UVP) merges the code from the hotfix branch on the master branch for archiving.

• The user (M+UVP) brings the content of the hotfix branch into the release branch such that they can be integrated in the future release.

• The user (M+UVP) brings the content of the hotfix branch into the devel branch such that the corrections can be integrated in the future release. Note that possible conflicts may exist on some pieces of code. They will have to be resolved before merging thus requiring the help from the other developers involved in the modifications.

• If needed, the user (M+UVP) deploys the code from hotfix branch in the dev environment, such that at this stage of the workflow, the same commit ID of the bioinformatics pipeline is deployed in the dev, valid and prod environment.

• The user (M+UVP) closes the GitLab issue and the GitLab issue that were previously opened.
4.2 Technical procedure

4.2.1 Step 1 - software development

Code writing

- The developer (D) reads the bug report or writes it in a new GitLab issue using the template `bug_report`.
- The developer (D) sets the local workspace on the hotfix branch, gets the last modifications from the remote repository and checks that the right branch is used:

```
git checkout hotfix
git status
 git pull
 git submodule update --init --recursive # if your git repo has submodules
```

- The developer (D) gets the commit ID that has been deployed in the prod environment. The commit ID is retrieved from the version file that is available in the folder in which the pipeline has been installed.

Important: It is essential that the commit ID that has been used to deploy the code is tracked in the install folder. For this purpose, the commit ID is stored in the version file.

- The developer (D) checks that the commit ID stored in the version file is the same as the commit ID of the hotfix branch from the local repository:

```
git branch -vv
```

- The developer (D) creates a local branch named `hotfix-id_commit-user`. The `id_commit` is the commit number of the version that has been deployed in the prod environment that needs to be fixed, and `user` is the unix login of the developer (e.g. hotfix-aa12bb34-phupe).

```
git checkout -b hotfix-id_commit-user
```

- The developer (D) implements the corrections, tests the modifications and commits them (see Code writing for the details about the command lines).
- The developer (D) temporarily pushes the new local branch on the remote repository:

```
git push origin hotfix-id_commit-user
```

- The developer (D) deploys the code from the `hotfix-id_commit-user` branch either in a personal environment for testing or in the dev environment to perform unit, integration, system and regression testing.
- The developer (D) checks that the bug has been corrected.

Deployment in the dev environment

The user (D+UD) deploys the bioinformatics pipeline in the dev environment from the `hotfix-id_commit-user` branch using the ad-hoc deployment scripts. The deployment is only based on a commit ID.

Danger: Tags with version number are not used to deploy the code but only a commit ID. Indeed, a tag can be easily removed or moved in git thus it is not a reliable information for tracking. This is why commit ID are used for deployment.
Testing

- The developer (D) checks that the bug has been fixed.

### 4.2.2 Step 2 - acceptance testing

#### Acceptance testing by end-users

- The user (D+UD) creates an GitLab issue using the template validation.
- The title of the GitLab issue must indicate the characteristics of the version to be validated.
- The description in the GitLab issue lists the new features/modifications that have to be communicated to the end-users.
- At the end of the description, a line such as **fyi: @user1, @user2, @user3** is added such that all the persons involved in the validation process receive a notification.

- The GitLab issue is labeled with validation.
- The GitLab issue is assigned to a user (M+UVP).
- The user (D+UD) or the user (M+UVP) sends an email to all the persons who are involved in the validation process.
- The end-users can start the acceptance testing process:
  - either the end-users validate the new release,
  - or the end-users do not validate the new release. Then, the reason are tracked in the GitLab issue that has been created. We go back to *Step 1 - software development*. The developer (D) develops the modifications requested by the end-users on a local hotfix-id_commit-user branch. The process is iterated until the validation by the end-users. The same GitLab issue is used to track all the information during the validation process until the final validation.

- Once validated by the end-user, the developer (D) creates a Create a Merge Request in Gitlab from the hotfix-id_commit-user branch on hotfix branch. The merge request is assigned to a user with the Maintainer role.
- The user (M+UVP) reviews and accepts the Merge Request.

#### Update the CHANGELOG

**Note:** The CHANGELOG file provides a simple history of the different versions of the bioinformatics pipeline. The version numbers are listed by decreasing order.

- A version number is added in the CHANGELOG using the following naming convention: version-x.y.z.
- Comments are added in the CHANGELOG to describe the most relevant functionalities added to the new release.

The CHANGELOG is divided into 3 sections:

- NEW FEATURES
- SIGNIFICANT USER-VISIBLE CHANGES
- BUG FIXES
Example of CHANGELOG file:

```
version-1.2.1

BUG FIXES
[BUGFIX] segfault corrected

NEW FEATURES
[MODIF] optimization of the sort function
[DOC] update of the end-user documentation

version-1.2.0

NEW FEATURES
[MODIF] use of GRCh38 reference genome
```

- The user (M+UVP) updates the hotfix branch to get the last modifications from the remote repository and checks that the right branch is used:

```
git checkout hotfix
git status
git pull
git branch -vv
```

- The user (M+UVP) updates the CHANGELOG. The version number is incremented using the following naming convention: version-x.y.z.
- If needed, the user (M+UVP) asks the other developers to define what comments should be added in the CHANGELOG and pushes the modifications on the remote repository:

```
git add CHANGELOG
git commit -m "[DOC] information about the version-1.2.4 after correction of the bug added in the CHANGELOG"
git push origin hotfix
```

### 4.2.3 Step 3 - check the installation process and new testing

#### Deployment in the valid environment

The user (M+UVP) deploys the pipeline in the valid environment from the hotfix branch using the ad-hoc deployment scripts. The deployment is only based on a commit ID.

**Danger:** Tags with version number are not used to deploy the code but only a commit ID. Indeed, a tag can be easily removed or moved in git thus it is not a reliable information for tracking. This is why commit ID are used for deployment.

#### Test the version deployed in the valid environment

The user (M+UVP) tests the bioinformatics pipeline.
Launch the operational testing in Jenkins

- The user (M+UVP) launches an analysis to generate the dataset that will be used as a reference.
- In the dedicated project in jenkins named with the suffix **_VALID**, the user (M+UVP) modifies the parameters of the operational testing (for example, a new reference dataset may be used for this purpose, etc.), the reference dataset will be the dataset that has been validated.
- The user (M+UVP) launches the operational testing.
- If the operational testing fails (the bioinformatics pipeline does not work or is not reproducible), go back to the *Development of corrections if needed*.

Development of corrections if needed

In most of the cases, the deployment in the **valid** environment is very simple and quick. However, it might be necessary to correct some bugs before the deployment in production. In that case:

- The developer (D) checkouts and updates the **hotfix** branch from the **local workspace**:

  ```
git checkout hotfix
git status
git pull
git branch -vv
  ```

- The developer (D) creates a local branch named **hotfix-id_version-user** (e.g. hotfix-version-1.2.3-phupe), the hotfix-version is retrieved from the CHAMELOG file) and uses it for the developments:

  ```
git checkout -b hotfix-id_version-user # change the id_version and user values
  ```

- The developer (D) implements the corrections, tests the modifications and commits them (see *Step 1 - software development* for the details about the command lines).
- The developer (D) temporarily pushes the local branch on the **remote repository**:

  ```
git push origin hotfix-id_version-user # change the id_version and user values
  ```

- The developer (D) deploys the code from the **hotfix-id_version-user** branch either in a personal environment for testing or in the **dev** environment to perform unit, integration, system and regression testing.
- Once the code validated, the developer (D) creates a *Create a Merge Request in Gitlab* from the **hotfix-id_version-user** branch on the **hotfix** branch using the template **merge_request_template.md**. The **Merge request** is assigned to a user with the **Maintainer** role.
- The user (M+UVP) reviews and accepts the **Merge Request**.
- The user (M+UVP) updates the **hotfix** branch from the **local workspace**:

  ```
git checkout hotfix
git status
git pull
git branch -vv
  ```

- The user (M+UVP) go back to *Deployment in the valid environment*. 

---

Chapter 4. Hotfix procedure
Add a tag with the version number

Once the new release has been validated and the installation in the valid environment is successful, the user (M+UVP) adds a tag (using the same version number that has been written in the CHANGELOG file) on the current HEAD:

```
tag-version4prod.sh -t version-1.2.3
git push --tags
```

**Note:** The script `tag-version4prod.sh` checks that the tag name is consistent with what was mentioned in the CHANGELOG and add the tag.

4.2.4 Step 4 - production deployment

Update the local repository with the version to deploy

It is likely that the local repository is not up-to-date anymore especially if a Merge Request has been submitted on GitLab. The user (M+UVP) updates the **local workspace**:

```
git checkout hotfix
git status
git pull
git branch -vv
```

- The user (M+UVP) creates a GitLab issue using the template `deploy_in_prod_hotfix`
  - The GitLab issue is labeled with production.
  - The GitLab issue number that has been used for the validation along with the GitLab issue number that describes the bug is added to the current GitLab issue.
  - The user (M+UVP) tracks all the steps that are performed for the deployment in the production environment (including link or name of datasets that are used).
  - The user (M+UVP) fills the GitLab issue at each step.

Deployment in the prod environment

The user (M+UVP) deploys the bioinformatics pipeline in the **prod** environment from the `hotfix` branch using the ad-hoc deployment scripts. **The deployment is only based on a commit ID.** The last commit ID from the **release** branch must be deployed.

**Danger:** Tags with version number are not used to deploy the code but only a commit ID. Indeed, a tag can be easily removed or moved in git thus it is not a reliable information for tracking. This is why commit ID are used for deployment.

Launch the operational testing in Jenkins

- The user (M+UVP) launches an analysis to generate the dataset that will be used as a reference.
• In the dedicated project in jenkins, the user (M+UVP) modifies the parameters of the operational testing (for example, a new reference dataset may be used for this purpose, etc.), the reference dataset will be the dataset that has been validated.

• The user (M+UVP) launches the operational testing.

• If the operational testing fails (the bioinformatics pipeline does not work or is not reproducible), go back to the Development of corrections if needed.

Bring the content of the hotfix branch into the master branch

• At this stage, there is a stable code on the hotfix branch that has been tested, validated and successfully installed in the prod environment.

• The user (M+UVP) checkouts and updates the master branch:

```bash
git checkout master
git status # everything must be cleaned
git pull
git branch -vv
```

• The user (M+UVP) brings the content of the hotfix into the master using the option `--no-ff` to avoid the fast-forward mode. This option will produce a new commit ID with a specific message to describe and track the merge:

```bash
git merge --no-ff hotfix```
# can be a bit verbose

git status # must be cleaned
git branch -vv
```

• The git status must absolutely says something like this (otherwise, ask for help before moving forward):

```bash
# On branch master
# Your **branch is ahead of 'origin/master' by** 113 commits.  
# (use "git push" to publish your local commits)
# 
# nothing to commit, working directory clean

# On branch master*
```

• The user (M+UVP) pushes the modifications on the remote repository:

```bash
git push origin master
```

Bring the content of the hotfix branch into the devel branch

• At this stage, there is a stable code on the hotfix branch that has been tested, validated and successfully installed in the prod environment and merged on the master branch.

• The user (M+UVP) checkouts and updates the devel branch:

```bash
git checkout devel
git status # must be cleaned otherwise, commit or stash your modifications
git pull
git branch -vv
```

• The user (M+UVP) brings the content of the hotfix branch into the devel branch:
• If the **devel** branch has been modified in the meantime, git will try to merge the modifications from the **hotfix** branch.

• If some files cannot be merged automatically, they will appear to have **conflicts** in the output of the `git status`:

```plaintext
# On branch devel
# You have unmerged paths.
# (fix conflicts and run "git commit")...
# (use "git add ..." to mark resolution)
# both modified:build.xml
```

• The conflicts have to be resolved manually. In that case, ask the help from the other developers.

• The files with resolved conflicts must be added to the staging area, committed, and the merge must be sent on the **remote repository**:

```
git push origin devel
```

**Bring the content of the hotfix branch into the release branch**

• At this stage, there is a stable code on the **hotfix** branch that has been tested, validated and successfully installed in the **prod** environment and merged with the **master** and the **devel** branches.

• The user (**M+UVP**) checkouts and update the **release** branch:

```
git checkout release
git status # must be cleaned otherwise, commit or stash your modifications
git pull
git branch -vv
```

• The user (**M+UVP**) brings the content of the **hotfix** branch into the **release** branch using the option **--ff** to use the fast-forward mode in order to have the exact same commit ID between the **release** branch and the **hotfix** branch. However this might not be always possible if the same pieces of code have been modified at the same time:

```
git merge --ff hotfix # may be verbose
git status # may say something
git branch -vv
```

• If the **release** branch has been modified in the meantime, git will try to merge the modifications from the **hotfix** branch.

• If some files cannot be merged automatically, they will appear to have **conflicts** in the output of the `git status`:

```plaintext
*# On branch release*
*# You have unmerged paths.*
*# (fix conflicts and run "git commit")...*
*# (use "git add ..." to mark resolution)*
*## both modified:build.xml*
```

• The conflicts have to be resolved manually. In that case, ask the help from the other developers.
• The files with resolved conflicts must be added to the staging area, committed, and the merge must be sent on the remote repository:

```bash
git push origin release
```

• The user (M+UVP) closes the GitLab issue and GitLab issue that have been opened.

**Back on the devel branch**

For security reason, the user (M+UVP) switches on the `devel` branch to avoid any risk of code modification on the `master` branch:

```bash
git checkout devel
git pull
git branch -vv
```

### 4.3 Graphical synopsis of the procedure
Step 1 - software development

If testing successful go to Step 2, if not go to Step 1.1

Step 2 - acceptance testing

If end-users validate, go to Step 2.1, if not go to Step 1.1

Step 3 - check the installation process and new testing

If testing successful, go to Step 3.7, if not go to Step 3.3

Step 4 - production deployment

If testing successful go to Step 4.3, if not go to Step 3.8
5.1 Git basics

This section describes some basics about git usage.

`git` considers two levels:

1. The remote repository
2. The local workspace specific to each developer that is located on the developer’s computer

5.1.1 Remote repository

The remote repository is accessible at GitLab.

5.1.2 Local workspace

First, the developers have to clone a project from the remote repository:

```
git clone git@gitlab.com:biogitflow/biogitflow-demo.git
```

It may ask you for your login and password but this can be avoided by Adding an SSH key to your GitLab account.

Organisation

The local workspace is divided into three areas:

1. The working tree is the tree with all the files that are visible with the standard `ls` command. This is where the developer will modify the code.
2. The staging area or index is where `git` stores the list of files that will be sent with the next `git commit` command.
3. the **commit area** contains the history of the commit (some of them maybe available only locally) from the current **HEAD**.

The command `git add` transfers a file from the **working tree** to the **staging area**.

**Tip:** All the files that are not supposed to be versionned can be listed in the `.gitignore` file.

The command `git commit -m "[MODIF] new algorithm added (Issue #10)"` transfers the files from the **staging area** to the **commit area**. So far, all the modified files are only present on the **local workspace**. We will see in what follows how to push them on the remote repository.

**Commands to cancel some actions**

In order to cancel the actions that have been performed in the **local workspace**, the command `git reset` is used with the following options:

1. `--soft` # deletes the files from the commit area
2. `--mixed` # deletes the files from the staging area (default)
3. `--hard` # reinitialize the **local workspace**

**Note:** The the `--mixed` option includes the `--soft` option, and the `--hard` option includes the `--mixed` option.

**Danger:** The `--hard` option can be very dangerous as you can permanently lose all your modifications from your **local workspace**.

Example of reset commands:

- `git reset --soft HEAD^` cancels the last commit
- `git reset --soft` is the inverse of `git commit` (while the commit has not been pushed on the remote repository)
- `git reset --mixed HEAD^` cancels the last (like `--soft`) and delete the files from the staging area
- `git reset --mixed` is the inverse of `git add`
- A typical use case of the command `git reset --mixed` is to delete a file that has been added by error in the staging area: `git reset --mixed my-file-not-to-be-committed`
- `git reset --hard HEAD^` resets all the files from the **local workspace** to the version before the current **HEAD** (some modifications might be lost)

**Note:** **HEAD** is the last commit of the **local workspace**

- **HEAD~5** is the 5th commit before **HEAD**
- **HEAD^** is equivalent to **HEAD~1**
5.1.3 Interaction between the local workspace and the remote repository

To get the last modifications from the remote repository into your \textit{local workspace}: \texttt{git pull}

To send your modification from your \textit{local workspace} into the remote repository: \texttt{git push}

5.1.4 Create tags

- list the tags
  \texttt{git tag}

- add a tag on the last commit (HEAD) in the \textit{local workspace}:
  \texttt{git tag version-1.0.0 -m "[TAG] version-1.0.0"}

- list the information about a tag:
  \texttt{git show version-1.0.0}

- send the tags into the remote repository:
  \texttt{git push --tags}

- delete a tag from the \textit{local workspace}:
  \texttt{git tag -d version-1.0.0}

- delete a tag from the remote repository:
  \texttt{git push origin :version-1.0.0}

- add a tag on a previous commit:
  \texttt{git tag version-1.0.0 -m "[TAG] version-1.0.0" hash_du_commit}

5.1.5 Configuration

- Identity:
  \texttt{git config --global user.name "<name>"}
  \texttt{git config --global user.email <email>}

- Default editor:
  \texttt{git config --global core.editor vim} (or any editor you prefer)

- List the aliases:
  \texttt{git config -l}

- Add an alias:
  \texttt{git config <scope> alias.<alias> <commande git>}
  - \texttt{ex: git config --global alias.co checkout}
  - \texttt{ex: git config --global alias.last 'log -1 HEAD'}

- The following scopes are available:
  --global
  --system
--local

5.1.6 Useful commands

- Initialize a git repository from an existing folder on your computer:
  ```
  cd <localdir>
  git init (creates the .git folder)
  git add <files>
  git commit -m "message"
  git remote add origin <url>
  git push -u origin master
  ```
- Information about the files in the local workspace:
  ```
  git status
  ```
- Difference between the local workspace and the master branch of the remote repository:
  ```
  git diff origin/master
  ```
- Difference between two commits for a file on the same branch:
  ```
  git diff HEAD^ <file>
  Add as many ^ as you want to step back (HEAD^^^)
  ```
- Difference between two specific commits:
  ```
  git diff <commit_1> <commit_2> <file>
  ```
- Information about the last commit in the commit area:
  ```
  git log
  git log --oneline
  ```
- Delete a branch from the local workspace:
  ```
  git branch -d mybranch
  ```
- Create a local branch named foo from the master branch of the remote repository:
  ```
  git checkout -b foo origin/master
  ```
- Create a local branch named bar that is not supposed to be pushed into the remote repository from the current branch:
  ```
  git checkout -b bar
  ```
- Information about the branches that are available in the local workspace:
  ```
  git branch -vv
  ```
- Send a local branch named foo in the local workspace into a branch named bar in the remote repository:
  ```
  git push origin foo:bar
  ```
- Delete the branch bar from the remote repository:
  ```
  git push origin :bar
  ```
- The syntax is generally `git push origin localname:remote`. When we want to remove something, just leave the localname empty
• Delete files that are not versioned in the **local workspace** (beware, you can lose data):
  
  ```
  git clean -n  # dry-run mode
  git clean -df
  ```

• Information about the URL of the remote repository:
  
  ```
  git remote -v
  ```

• Modify the URL of the remote repository:
  
  ```
  git remove set-url origin ssh://git@gitlab.com/project.git
  ```

• Visualize the commit history in an interface:
  
  ```
  gitk
  ```

**Temporary shelf some modifications**

Imagine that you modified some files but the modifications are not yet completed to be committed. In the meantime, you have to correct some bug on the **hotfix** branch. The command `stash` allows you to store your modifications. Otherwise, you will not be able to checkout the **hotfix** branch.

```

  git stash stashes the current modifications
  git stash list lists the existing stashes
  git stash show my_stash_id details information about a stash
  git stash apply my_stash_id restores the modifications from a stash
  git stash drop my_stash_id deletes a stash

```

**Rollback**

Imagine that some modifications have been pushed on the remote repository but they should not have been pushed. Therefore, we have to restore the repository at its previous state:

• go back to the last valid commit with the common ancestor:
  
  ```
  git reset --hard 5a15e6c26300bb74cf95f9f4b33b6a7288d67524
  ```

• apply the valid commit between the latest commit and the common ancestor:
  
  ```
  git cherry-pick 752bbf07d5c6f3998a1a9679e02b1ef015b301f
  ```

• push the modifications to the remote repository:
  
  ```
  git push -f origin devel
  ```

However, the `git pull` will not erase the commits that have been deleted from the remote repository if they are still present in the **local workspace**:

```

  git pull
  git status
  ```

If the `git status` outputs a message like **Your branch is ahead of ‘origin/devel’ by X commit**, it means that the commits are still present on your **local workspace**. Dot not push anything but:
• either delete the **devel** branch and recreate it from the remote repository:
  
git checkout -b revert_backup  
git branch -d devel  
git checkout -b devel origin/devel

• or force the deletion of unnecessary commits:
  
git reset --hard origin/devel

### 5.1.7 Git and Gitlab

GitLab is a graphical interface that allows the managements of the projects inside the remote repository.

### 5.1.8 Track the Issue in the commit message

GitLab offers a functionality to report Issues. The ID of an Issue can be added in a commit message for better tracking of the modifications:

```
git -m "[BUG] bug correction (Issue #10)"
```

This way, the information regarding the commit is tracked directly in the Issue.

### 5.1.9 Annexes

For more information visit https://www.atlassian.com/git

### 5.2 Git advanced

#### 5.2.1 Merge 2 branches from different repositories

How to retrieve code from the source_repo_branch of the source_repo project onto the target_repo_branch of the target_repo:

```
git clone https://gitlab.com/source_repo  
git clone https://gitlab.com/target_repo  
cd target_repo  
git checkout target_repo_branch  
git remote add source_repo ..../source_repo  
git fetch source_repo  
git checkout -b merge-source-target  
git merge source_repo/branche_source_repo
```

At this stage, resolve possible conflicts and commit and then:

```
git checkout target_repo_branch  
git merge merge-source-target  
git remote rm source_repo
```
5.2.2 Add a submodule in a repository

If you want to create a submodule, you can edit and modify the variables in the file `createSubmodule.bash` and follow the procedure.

5.2.3 Generate some statistics about the repository

Number of commits by user:

```
git shortlog -s -n --since "JUN 30 2018"
```

5.3 Naming convention for the commit messages

A prefix among the following is added in the commit message:

- `[MODIF]`: modify features or add new ones
- `[BUGFIX]`: bug correction
- `[MERGE]`: merge branches
- `[DOC]`: add/modify documentations
- `[TAG]`: add a tag
- `[ADD]`: add new files
- `[DEL]`: delete files

In the commit message, the number of the Issue related to the modifications is tracked as follows: “[MODIF] Add star mapper (Issue #11)”.
6.1 Create a new project in Gitlab

We recommend that the project is created from the biogitflow template or you can follow the procedure below.

6.1.1 Add the new project

- Select Projects in the upper left menu
- Click on New project
- Link your project to a group (if needed)
- Give a name and a description to your new project
- Check that the Visibility Level is set to Private
- Select the option Initialize repository with a README

6.1.2 Add the templates for the issues and merge requests

- Clone the new project:

```
git clone git@gitlab.com:biogitflow/biogitflow-demo.git
cd biogitflow-demo
```
• Create the hidden folder `.gitlab`:

```bash
mkdir .gitlab
```

• Copy the content of the directory `biogitflow templates` such that the `.gitlab` folder contains the following folders:

```bash
issue_templates
merge_request_templates
```

• Push the templates on the `master` branch on the remote repository:

```bash
git add .gitlab
git commit -m "[ADD] templates for issues and merge requests"
git push origin master
```

### 6.1.3 Create the branches

• Click in the `Repository` menu in the left panel, in order to create the `release`, `hotfix` and `devel` branches from the `master` branch
  
  – Click on the `New branch` button:

  ```bash
  branch = devel
  branch = master
  ```

  – in the form, fill `Branch name = devel` and `Create from = master`
– Iterate the same procedure twice to create:

– the release branch

– the hotfix branch

### 6.1.4 Set the default branch

- Click in the Settings menu in the left panel, select the Repository sub-menu, then click on the Default Branch button to expand the section:

  ![Default Branch Settings](image)

- Select Default branch = devel in the scrolling menu and click on the Save changes button:

### 6.1.5 Set the protected branches

- Click in the Settings menu in the left panel, select the Repository sub-menu, then click on the Protected Branches button to expand the section:
• Protect the release branch such that only the users with the Maintainers role can push and merge:
  – In the form select Branch = release, Allowed to merge = Maintainers and Allowed to push = Maintainers, then click on the Protect button:
• Check that master, hotfix and release branches are protected

### 6.1.6 Set the protected tags

The use of the tags is reserved to the user with the Maintainer role.

• Click in the Settings menu in the left panel, select the Repository sub-menu, then click on the Protected Tags button to expand the section:
• In the fields Tag, add * and then click on Create wildcard *
• Check that the field Allowed to create is set to Maintainers and then click on the Protect button:

### 6.1.7 Add a set of labels for the issues

• Click in the Issues menu in the left panel, select the Labels sub-menu
• It is possible that some labels already exist if the project has been created inside a group for which some labels have been already set (using the Promote to Group Label functionality for example)
• If there is no label, then click on Generate a default set of labels:
• Create the labels need by biogitflow documentation that are not in the default set:
6.1. Create a new project in Gitlab
Chapter 6. GitLab tutorial
6.1. Create a new project in Gitlab
• Create the new label **validation** for the Validation of a new release by the end-users:

• Create the new label **production** for the Deployment in the production environment:

• Check that all the labels required by biogitflow documentation are available:
6.1.8 Add members to the project

- Click in the Settings menu in the left panel, select the Members sub-menu
  - by default, all the members of the group have access to the new project
  - if needed, add additional members with their specific role

6.2 Create a Merge Request in Gitlab

6.2.1 Add a new Merge Request

- Click in the Merge Requests menu in the left panel:
6.2.2 Set the source branch

New Merge Request

Source branch

- phupe/test-phupe
- hotfix-0ebf8567-phupe

Target branch

- phupe/test-phupe
- hotfix-0ebf8567-phupe

Merge branch 'hotfix' into 'master'

- phupe authored 26 minutes ago
- phupe authored 3 days ago

Compare branches and continue
6.2.3 Set the target branch

New Merge Request

Source branch

phupe/test-phupe  
hotfix-0ebf8567-phupe

[BUGFIX] correction de l'anomalie
phupe authored 15 minutes ago

Target branch

phupe/test-phupe  
hotfix

Select target branch

search branches

✔ hotfix
  hotfix-0ebf8567-phupe
  master
  release
  devel
  valid

Compare branches and continue

6.2.4 Add a description using the Merge Request template

6.2.5 Assign the Merge Request to a user with the Maintainer role

Assignee

Assign to me

6.2.6 Add a label

The default label in the Merge Request template is . Change it if needed:
6.2.7 Set the option to delete the branch after the merge

Source branch: hotfix-0ebf8567-phupe
Target branch: hotfix

- Remove source branch when merge request is accepted.
6.2.8 Submit the Merge Request

Click on Submit merge request.

6.2.9 Review and merge of the Merge Request by the Maintainer

The Maintainer reviews the Merge Request and click on Merge:

- Merge
- Delete source branch
- Squash commits

- 2 commits and 1 merge commit will be added to hotfix. Modify merge commit

You can merge this merge request manually using the command line
CHAPTER 7

Configure an operational testing

This section describes how to configure an operational testing with jenkins.

Note: Other tools can be used for this purpose such as the he GitLab CI/CD.

7.1 Add a new project

- Connect on the jenkins web interface.
- Create a new operational testing by clicking on the New Item button

7.2 Set the periodicity of the testing

- Define the periodicity to launch the operational testing:
7.3 Write the command line to execute the testing

- Select the Execute Shell in the scrolling menu to add a build step:

```
/bioinfo/pipelines/foobar/prod/test/run-test-op.sh
```

Would last have run at Wednesday, August 30, 2017 5:59:27 AM CEST; would next run at Thursday, August 31, 2017 5:59:27 AM CEST.
7.4 Notify the end-users if the testing fails

- Add a post-build step such that the end-users are notified if the bioinformatics pipeline fails during the operational testing:

- Add the emails to be notified:

  ![Email Notification](image)

  - [ ] Send e-mail for every unstable build
  - [ ] Send separate e-mails to individuals who broke the build
CHAPTER 8

Indices and tables

- genindex
- modindex
- search