ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

03
Chapter

Workflows with Git

2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

"Git devient plus facile une fois que vous avez compris l'idée de base selon laquelle les branches sont des endofoncteurs homéomorphes mappant des sous-variétés d'un espace de Hilbert"

Isaac Wolkerstorfer

Outline

1

Collaborating With Git

Workflow

- A workflow defines a plan for organizing the collaborative work of multiple developers for the production of software.
- Workflow based on Git
  - Everything goes through branches
- Multiple workflows
  - GitHub Workflow
  - Git Workflow
  - GitLab Workflow
- Suppose 2 types of collaborators using the forge GitLab
  - Developer
  - Version maintainer

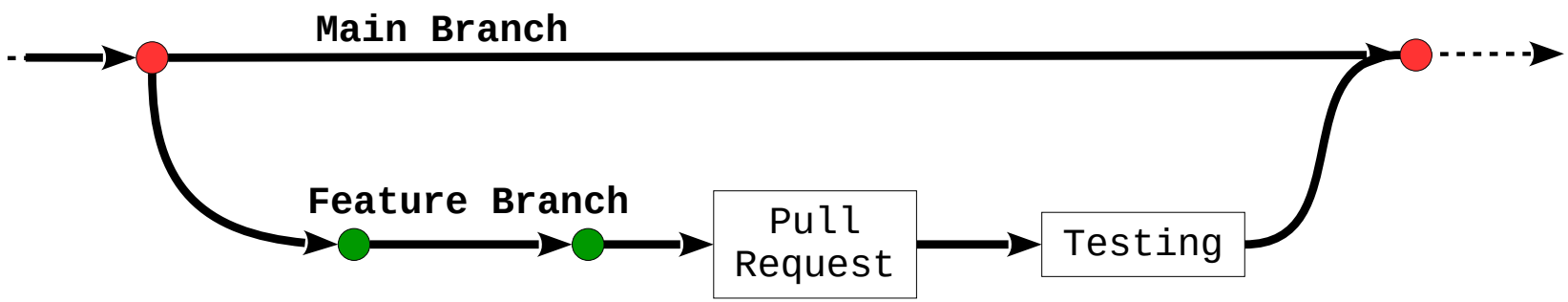
Outline

1
Collaborating With Git

2
GitHub Workflow

GitHub Workflow

- The simplest: suitable for small projects like student project
- Branches: Only 2 types of branch
  - **main**: **Permanent** production branch. Contains production-ready code.
  - ***x-feature***: **Temporary** development branch, where *x* is the issue number and *feature* is the issue name.



The “GitHub” Workflow

- **Developer**

1. Assign an issue
2. Run `git fetch` on the command line
3. Create a new branch `x-feature`
4. Edit code, `git commit` …. etc.
5. git pull origin main
6. Resolve conflicts
7. Push on GitLab
8. Create merge request

- **Version maintainer**

9. Assign a reviewer

1. Developer

10. Fix code review comments
11. Deal with any pipeline failures

2. Version maintainer

12. Accepts and merges, then the issue is automatically closed by GitLab and the branch is deleted on the remote repository

3. Developer

13. Delete branch on the local repository

Outline

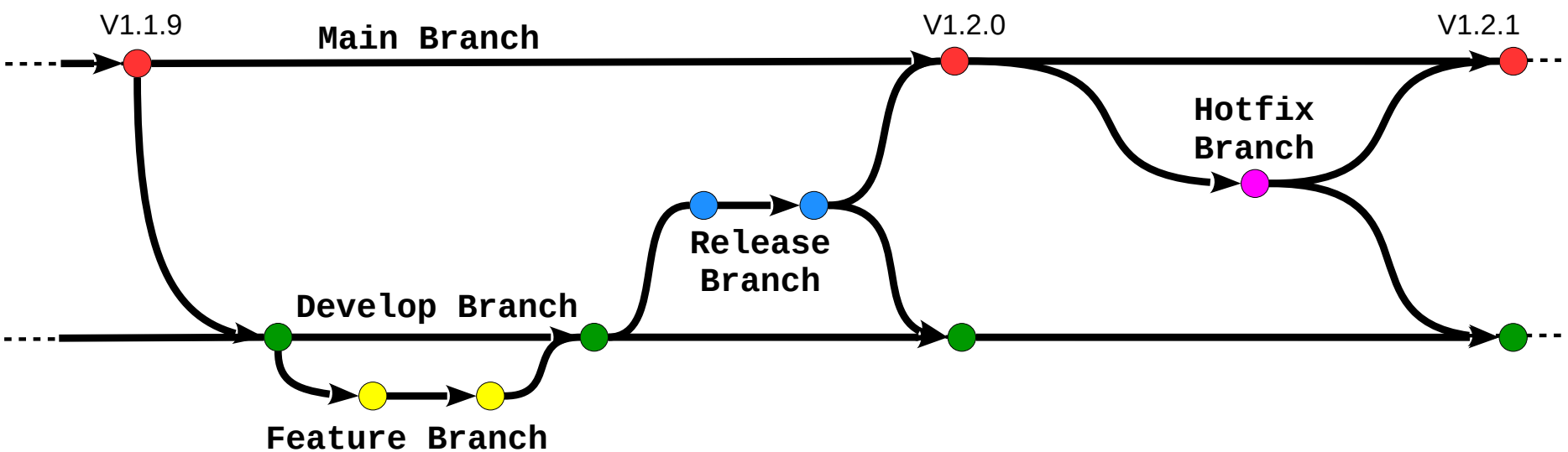
1
Collaborating With Git

2
GitHub Workflow

3
Git Workflow

Git Workflow

- The most complete: suitable for large projects
- Branches
  - **main**: **Permanent** production branch (stable version of the software)
  - **develop**: **Permanent** integration branch
  - **feature/x-ff**: Temporary feature development branch (where *x* is the issue number)
  - **hotfix/x-*issue***: Temporary bug fix branch
  - **release/*x.y.z***: Temporary production release branch



The “GitHub” Workflow

- **Developer**

1. Assign an issue
2. Run `git fetch` on the command line
3. Create a new branch `x-feature`
4. Edit code, `git commit` …. etc.
5. git pull origin main
6. Resolve conflicts
7. Push on GitLab
8. Create merge request

- **Version maintainer**

9. Assign a reviewer

1. **Developer**

10. Fix code review comments
11. Deal with any pipeline failures

2. **Version maintainer**

12. Accepts and merges, then the issue is automatically closed by GitLab and the branch is deleted on the remote repository

3. **Developer**

13. Delete branch on the local repository

Outline

1
Collaborating With Git

2
GitHub Workflow

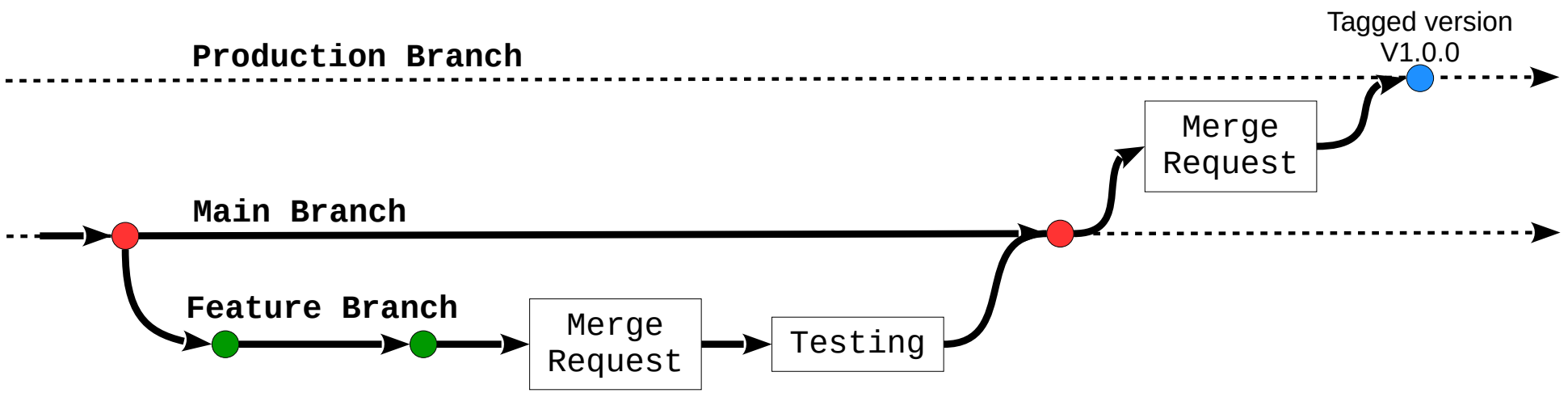
3
Git Workflow

4
GitLab Workflow

GitLab Workflow (basic version)

- GitLab Git Workflow is a branching strategy that integrates with GitLab's CI/CD pipelines to systematically build, test, and deploy code changes.
- Branches
  - **main**: Permane h
  - **production**: Pe anch
  - ***x-feature***: Tem anch (where x is the issue number)

Ajouter :
1/issue
2/ Create merge request



GitLab Workflow

- With GitLab Flow, all features and fixes go to the main branch while enabling production and stable branches.
- GitLab Flow incorporates a pre-production branch to make bug fixes before merging changes back to main before going to production. Teams can add as many pre-production branches as needed — for example, from main to test, from test to acceptance, and from acceptance to production.
- Workflow
- Main → x-feature (x-hotfix) → pull, push → merge request → code review, discussion, approval → production → main
- Start with an issue
- Create a branch from this issue inside of the web interface
- Copy/paste the branch name and pull that branch locally
- Edit the code
- Commit & Push to gitlab
- Create a merge request
- Accept the merge request

CI/CD

- Continuous integration
- Continuous deployment
- Continuous delivery
-
-
- Pipeline → jobs

GitLab Workflow

- 1. Open a GitLab issue (or assign yourself to one)
- 2. Create a branch
- 3. Develop in the new branch
- 4. Open a merge request and ask for code review
- 5. Rebase branch and solve conflicts
- 6. Merge the branch into main
- 7. Delete the branch
- 8. Close the GitLab issue

- Create an issue
- Create a merge request in the issue with the Create merge request button
- Run git fetch on the command line
- git checkout MR-BRANCH-NAME
- Edit code, git commit …. etc.
- git push MR-BRANCH-NAME
- Deal with any pipeline failures…
- Assign someone to review the MR
- Respond to review issues, etc.
- The reviewer accepts and merges, then the issue is automatically closed by GitLab

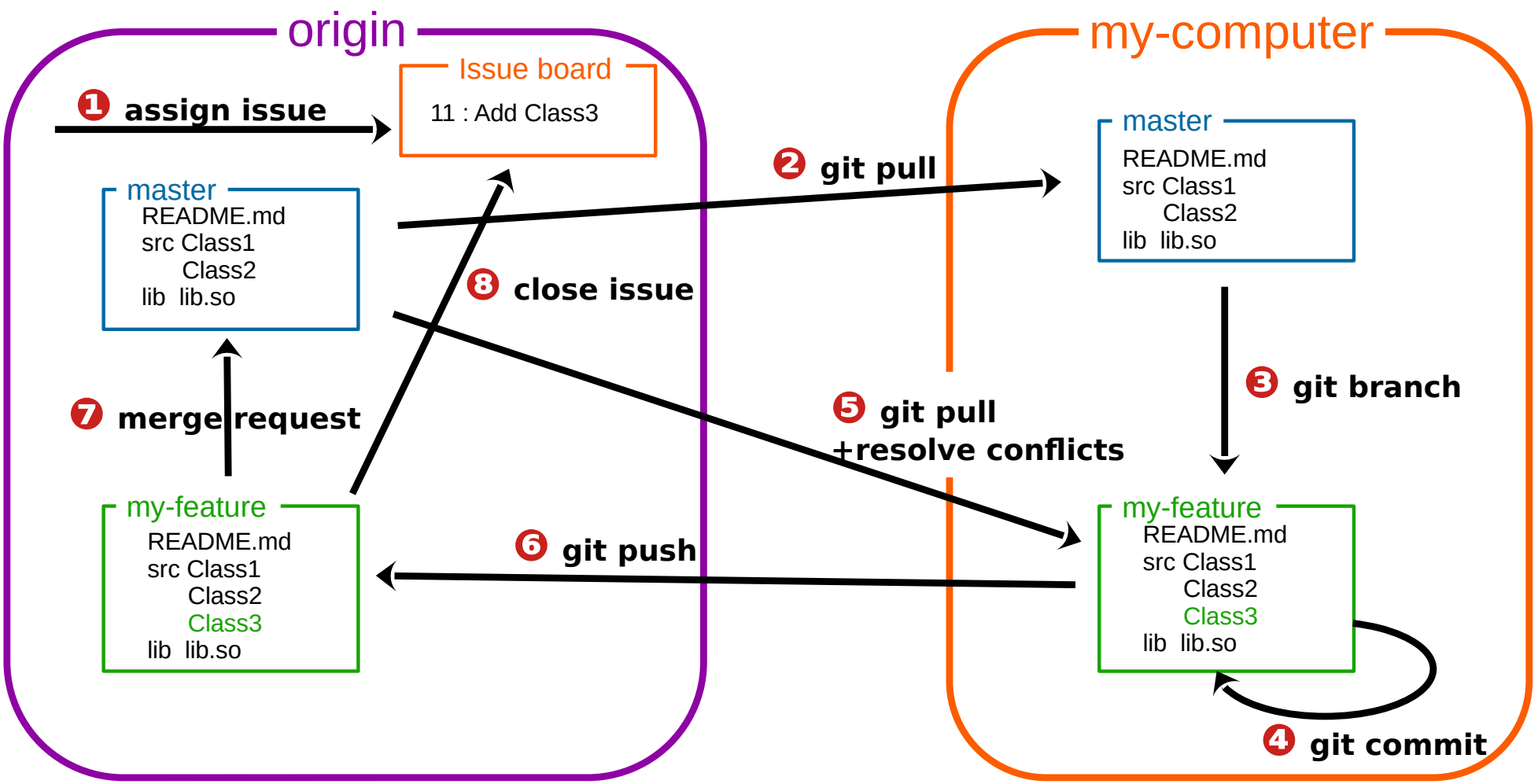
Developer Workflow (1)

- A developer adds a feature to the software
  - Get the last version of the `main` branch from the remote repository
    - `git pull origin main`
  1. Create a new branch (eg query-database)
     - `git branch query-database`
     - `git checkout query-database`
  2. Develop the feature
     - Edit / compile / test…
  3. At the end of the day (/lab), push the branch to the remote repository (for backup purpose)
     - `git add .`
     - `git commit -m "message"`
     - `git push`

Developer Workflow (2)

- At the end of the feature development
  1. Get the last version of the `main` branch from the remote repository
     ```
     1.git pull origin main
     ```
  2. Merge the branch in `main` and solve the conflicts
     ```
     1.git checkout query-database
     2.git merge main
     ```
  3. Execute all the tests
  4. Push the branch to the remote repository
     ```
     1.git add .
     2.git commit -m "message"
     3.git push
     ```
  5. On GitLab, use the UI to create a « merge request »

The GitLab Workflow



Outline

1
Collaborating With Git

2
GitHub Workflow

3
Git Workflow

4
GitLab Workflow

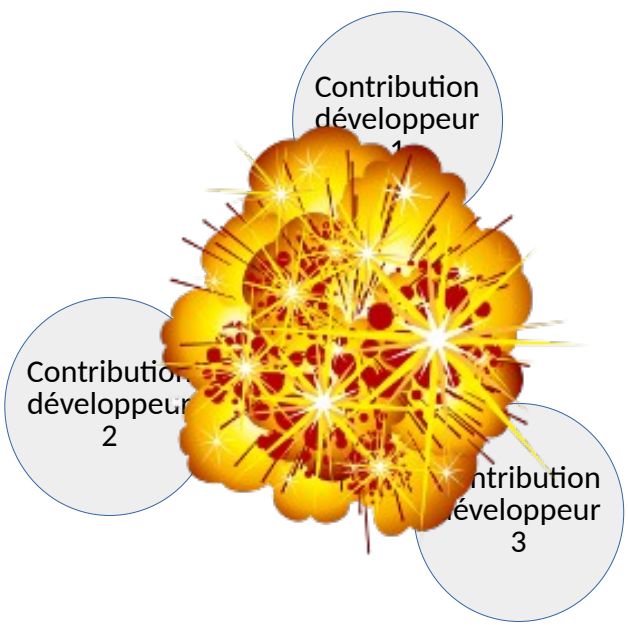
5
Best Practices

Best practices

- One branch = one functionality / hotfix
  - Develop using small steps to avoid integration problems and be able to go back to previous versions, for example:
- Short-lived branches
  - A branch must be short-lived, at most 1 lab session
  - Old branch becomes difficult to merge since it diverges more and more from the main branch
  - Create a new branch for each new contribution
  - Long-lived branch : master, develop
  - Delete branch after merged
- Small commit
  - Develop using small steps to avoid integration problems and be able to go back to previous versions, for example:
  - one *commit* every 20 minutes.
  - one *merge request* per day.
  - Keep atomic commit (one thing)

Best practices

- Continuous CI/CD
  - Automate code test and build checking
  - Never merge into main code which does not compile.
- Avoid big bang integration
  - Daily **continuous integration**



Semantic Versioning

- Version number: **MAJOR.MINOR.PATCH**
- Increment the:
  - MAJOR version when you make **incompatible changes**
  - MINOR version when you **add functionality** in a backward compatible manner
  - PATCH version when you make backward compatible **bug fixes**
- Benefit
  - Suppose there is a function library called “Fire Truck”, which requires another suite named “Ladder”.
  - When the fire engine was created, the ladder version number was 3.1.0.
  - Because fire trucks use some version of the new features of 3.1.0, you can safely specify that the version number of the ladder is equal to 3.1 0 but less than 4.0.0.
  - In this way, when ladder versions 3.1.1 and 3.2.0 are released, you can incorporate them directly into your suite management system because they can be compatible with pre-existing software dependencies.

Conventional Commits

- Format:

```
<type>(scope): <subject>

[optional body]

[optional footer]
```

- Example

```
feat(browser): add onUrlChange event

Add new event to browser:
- forward popstate event if available
- forward hashchange event if popstate not available
- do polling when neither popstate nor hashchange available

Closes #392
```

Conventional Commits

- Scope
- Subject :
  - impératif présent e.g. *add* (not *adds*)
  - Pas de majuscule au début
  - pas de point à la fin
  - Pied de page: BREAKING CHANGE → MAJOR (annoncé avec !)

Conventional Commits

- Types: (link to semantic versioning)
  - **feat**: add a new feature (MINOR)
  - **fix**: a bug fix (PATCH)
  - **docs**: changes on documentation (MINOR)
  - **style**: changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc) (PATCH)
  - **refactor**: a code change that neither fixes a bug nor adds a feature (PATCH)
  - **perf**: a code change that improves performance (PATCH)
  - **test**: adding missing tests or correcting existing tests (PATCH)
  - **build**: changes that affect the build system or external dependencies (MINOR/MAJOR)
  - **ci**: changes to CI/CD configuration files and scripts (PATCH)
  - **chore**: other changes that don't modify src or test files (PATCH)
  - **revert**: reverts a previous commit (MAJOR/MINOR)
  - !: breaking change (MAJOR)

Outline

1
Collaborating With Git

2
GitHub Workflow

3
Git Workflow

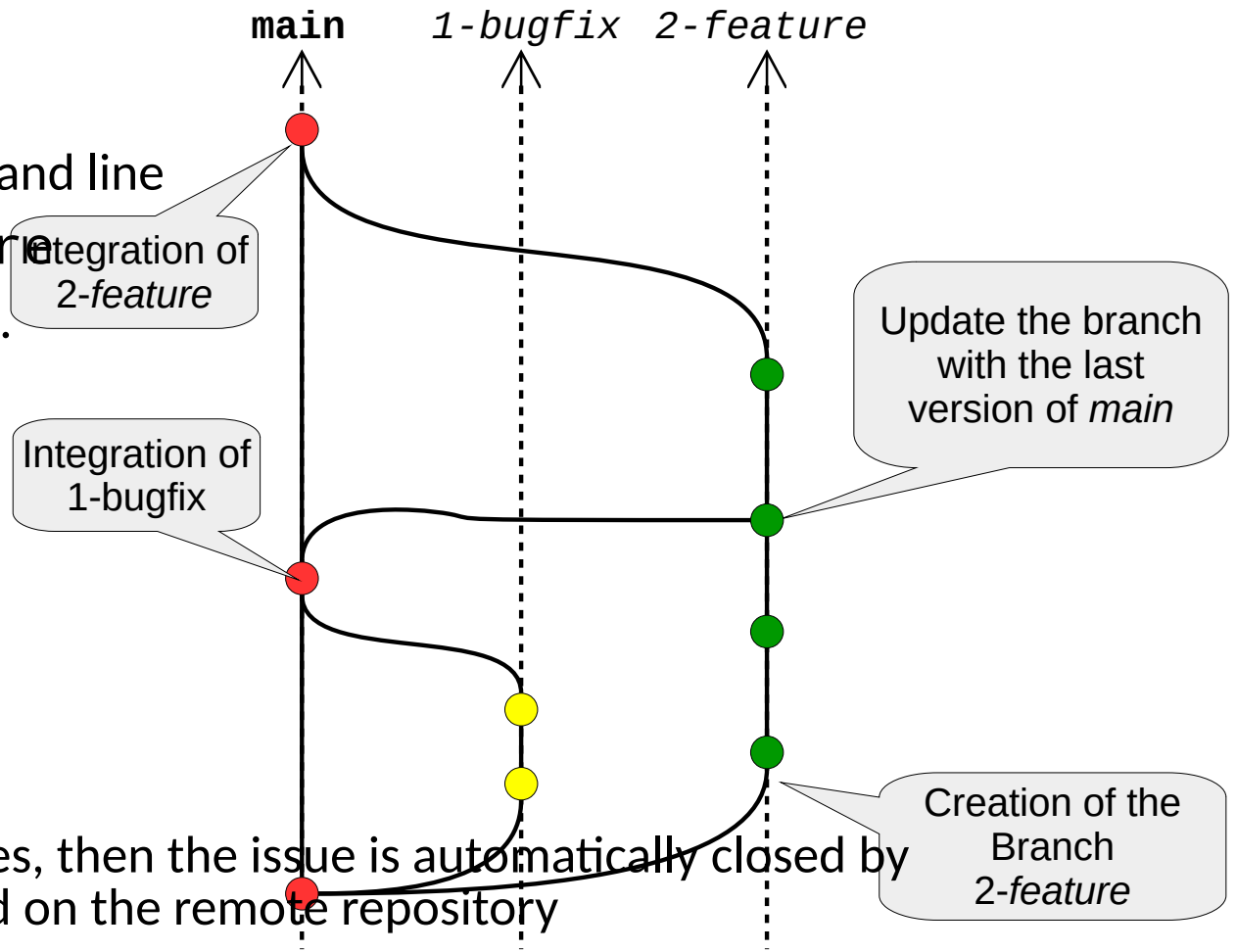
4
GitLab Workflow

5
Best Practices

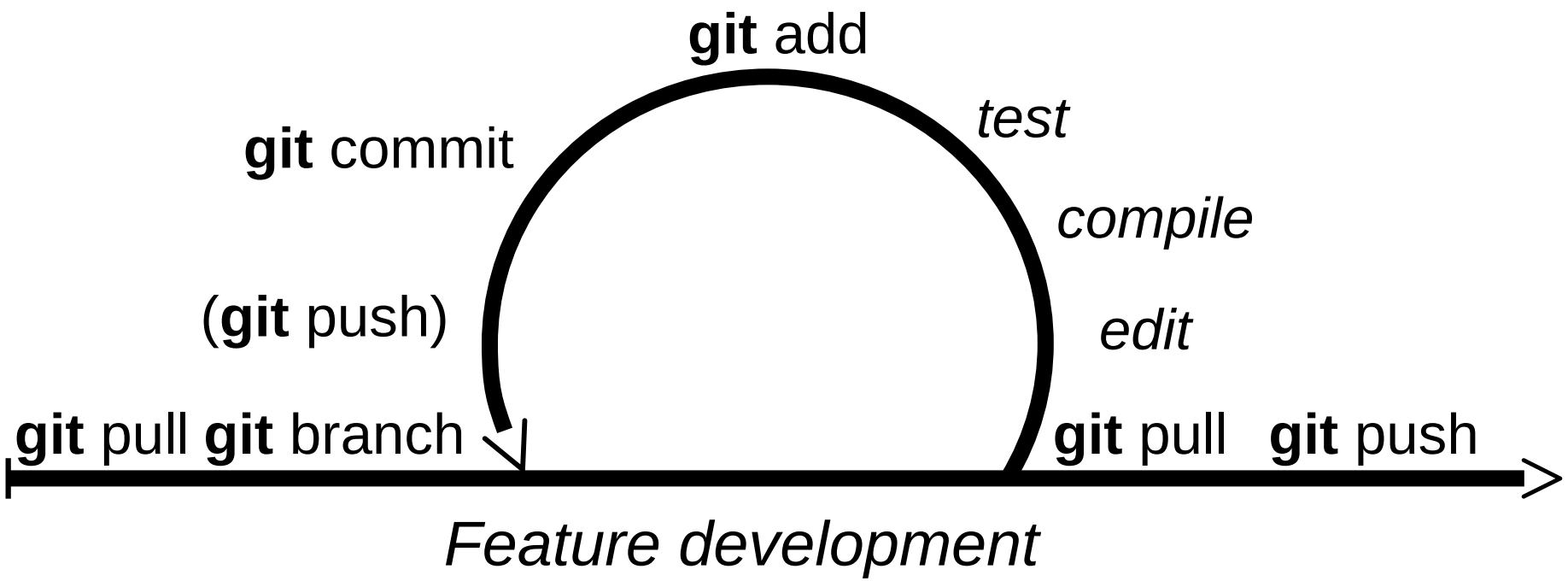
Demo
GitLab Workflow

The “GitHub” Workflow

- Developer
  1. Assign an issue
  2. Run `git fetch` on the command line
  3. Create a new branch `x-feature`
  4. Edit code, `git commit` …. etc.
  5. git pull origin main
  6. Resolve conflicts
  7. Push on GitLab
  8. Create merge request
- Version maintainer
  - Review
  - Deal with any pipeline failures…
  - The reviewer accepts and merges, then the issue is automatically closed by GitLab and the branch is deleted on the remote repository
- Developer
  - Delete branch on the local repository



TL;DR


git add
test
compile
edit
git pull **git** push
git pull **git** branch
(**git** push)
git commit
Feature development

Git commands

- Developer

  `$ git pull origin main`

  `$ git branch` *`x-feature`* (where x is the issue number)

  `$ git checkout` *`x-feature`*

  - Development : edit / compile / test

    `$ git commit -ma “message”`

    `$ git pull origin main`

  - Resolve conflict

    `$ git pull`

  - Create a pull request