Git

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction ... 2
- What is revision control? ... 3

Using Git ... 4
- Core Operations ... 5
- Local Workflow ... 6
- Remote Workflow ... 7
- Commands ... 9
- Using GitLab ... 10

Branching ... 12
- What is branching? ... 13
- What are feature branches? ... 14

Branching Models ... 16
- Trunk-Development ... 17
- Git Flow Development ... 19

Bibliography ... 20

Introduction

What is revision control?

Version Control Systems (VCS) are software systems that track changes to a set of related files. e.g., Git, Subversion, Perforce [1].

There are significant benefits to version control:

- **Resiliency**: source code is tracked by the system, and not isolated on an individual's computer. This reduces the risk of losing valuable source code.
- **Versioning**: the ability to manage sets of files together. This supports operations on related files. e.g., comparing your current code with the previously working version to identify an issue.
- **Collaboration**: a VCS provides the necessary capabilities for multiple people to work on the same code simultaneously, while keeping their individual changes isolated.

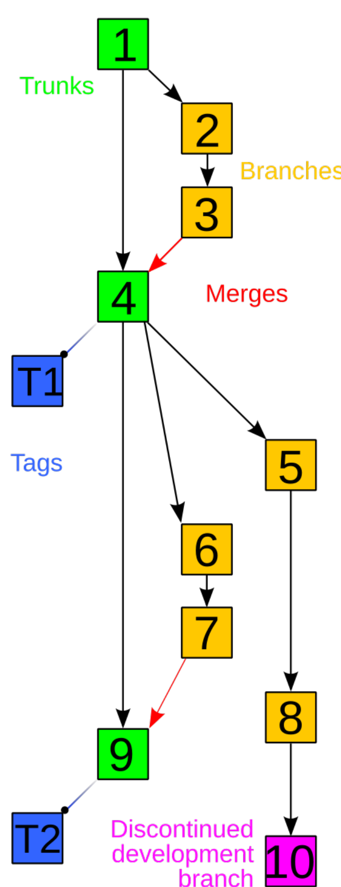

Figure 1: Example history graph of a revision-controlled project.

Using Git

Core Operations

Git is designed around these core concepts:

- **Repository**: The location of the canonical version of your source code ("Local Repo" in this diagram).
- **Working Directory**: A local copy of your repository, where you will make changes before saving them in the repository.
- **Staging Area**: A logical collection of changes from the working directory that you want to collect and work on together (e.g. a feature that resulted in changes to multiple files). You move files to the staging area prior to pushing them to the repo.

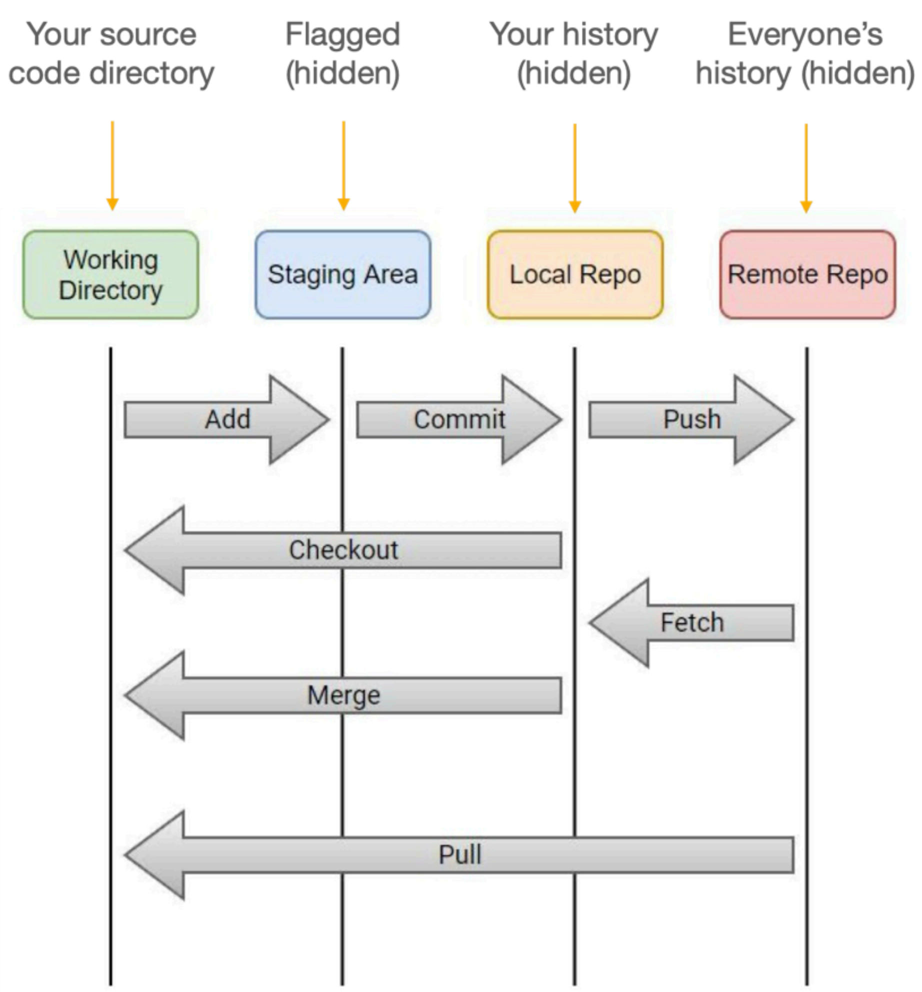

Figure 2: Git operates by using a series of operations to move a set of files between between your local system and the repository.

Local Workflow

A local Git workflow looks like this:

1. Create a project directory.
2. Initialize a git repository in this directory using `git init`. This creates a local git repository and allows you to start tracking changes.
3. Make changes to your source code in your favourite editor e.g., modify, add or delete files.
4. Add the changed files to your staging area using `git add`.
5. Commit from the staging area to save to the local repository with `git commit`.

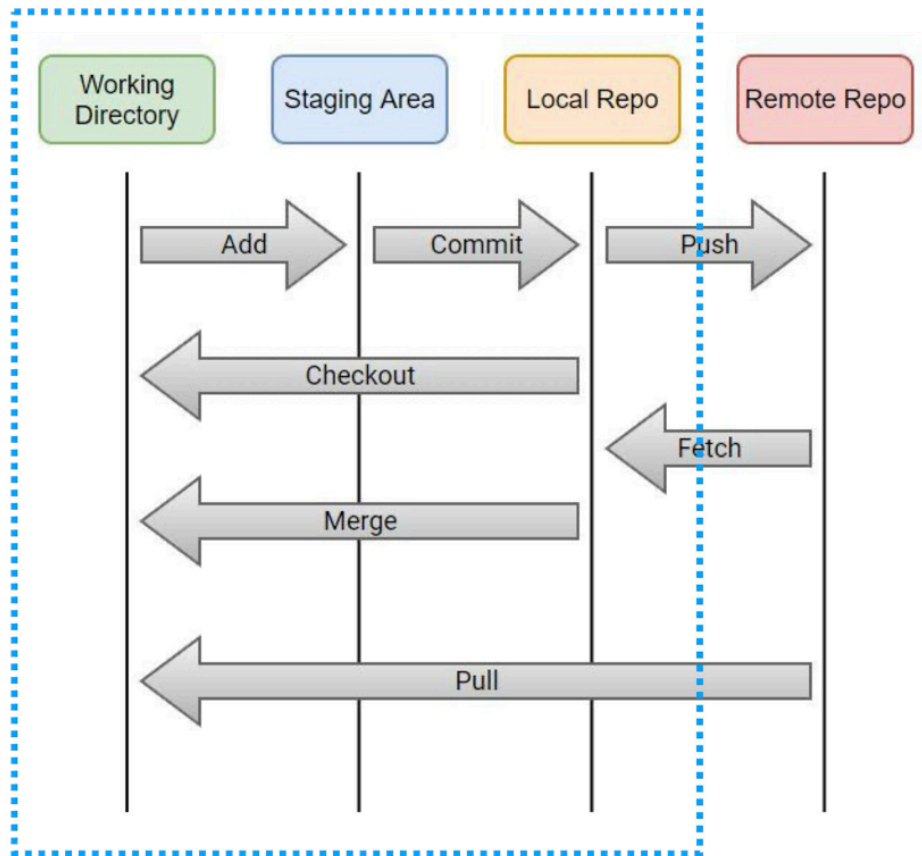

Figure 3: You can work entirely on your own computer, without accessing the remote repository… at least for a while.

Remote Workflow

You can use Git locally without restrictions. However, we often want a remote repository that contains everyone's changes.

- It helps us coordinate changes.
- It provides a single "source of truth" which can be backed-up.

To work with a remote repository:

- Setup a remote repository, or use an existing hosting site (e.g. GitLab).
- Add a connection between local and remote repositories.
- After committing making local changes, "push" them to the remote repository (`git push`).

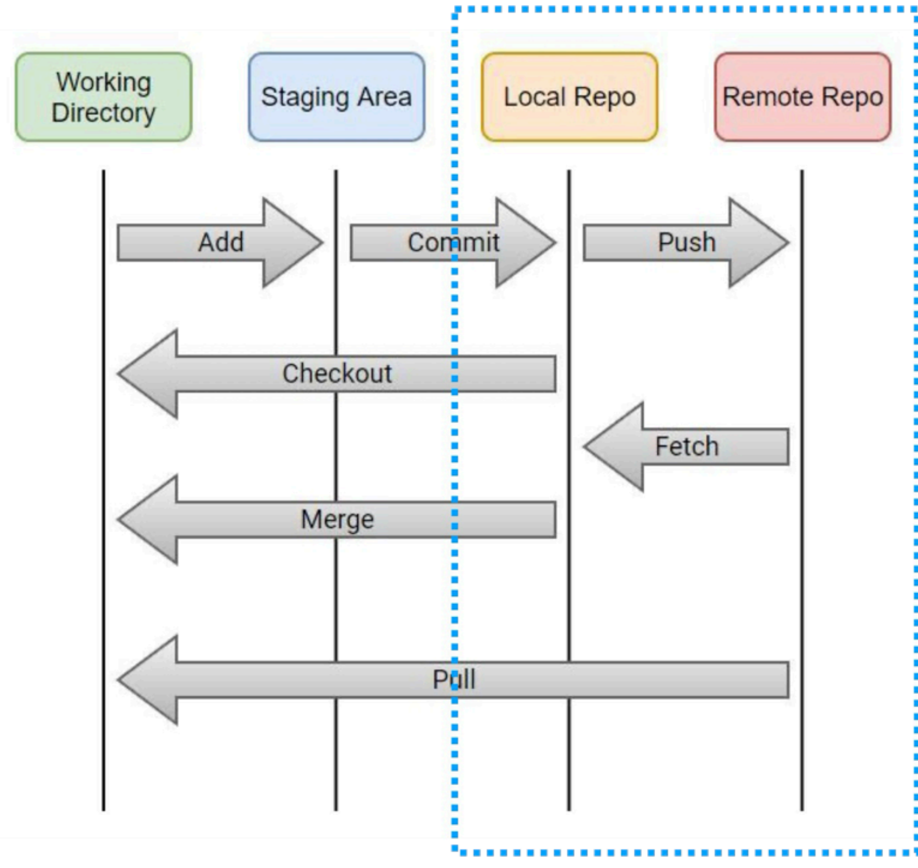

Figure 4: You normally will push your history from a local repo to a shared, remote repo. This allows other developers to see your changes.

Remote Workflow

The common workflow for working with a remote repository is:

1. Initialize a git repository in the remote directory e.g. in GitLab when you create a new project.
2. Clone the remote repository to create a local project directory on your computer i.e. `git clone` using the URL of the repo that you created in the previous step).
3. Make changes to your source code in your favourite editor.
4. Add the changed files to your staging area using `git add`. Commit the files in the staging area to save to the repository `git commit`. This two-step process ensures that these files are tracked and versioned as a single change.
5. Push the changes from your local repo to the remote repo using `git push`.
6. Check that your changes have been saved by using `git status`.

Commands

Command	Description	Example
git init	Create a new repository in the current directory.	$ mkdir repo; cd repo $ git init
git add	Add a file to the staging area	$ git add readme.md
git commit	Commit all staged files.	$ git commit -m “Added readme”
git status	Display current status.	$ git status
git checkout	Checking a specific commit to this working directory. Use to revert a file.	$ git checkout main.kt
git clone	Create a local copy of repository.	$ git clone https:/…
git pull	Merge changes into the local repository, usually before committing	$ git pull
git remote	Modify the remote connection	$ git remote -v

Using GitLab

Setup GitLab

1. Create your project in GitLab.
2. Clone the project URL from the project home page. This gives you a working copy. e.g.,

```
git clone https://git.uwaterloo.ca/cs346/demos/laws-of-se.git
```

3. Add your source code to that directory, commit and push it.

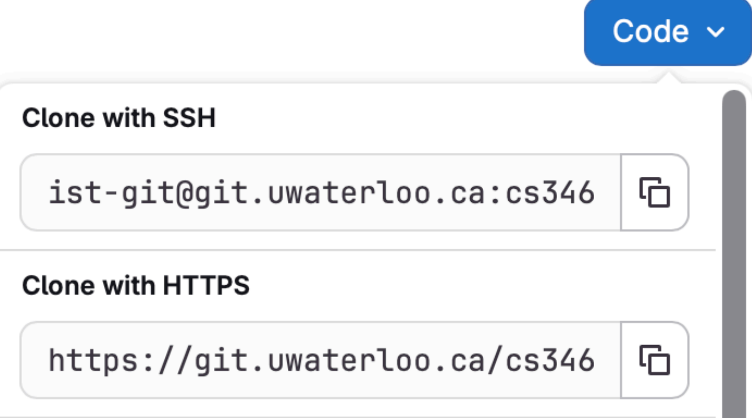

Figure 5: In your project, click the `Code` button to reveal the URI of your project.

GitLab Workflow

The simplest team workflow you can use is this:

1. Everyone has a local copy of the remote repository.
2. You complete your local changes, and have them working properly.
3. Use `git pull` to merge in any changes that other people have made since you started your work. You may need to manually resolve merge conflicts.
4. Next, `git add` any files that have changed, and `git commit` with a meaningful message.
5. Finally, `git push` your changes to the remote repository.

This works. However, it works best when there is no possibility of a merge conflict e.g., you are the single developer, or it's a trivial change.

- We want to reduce the risk of merge conflicts.
- We'll use branches to minimize this risk.

Branching

What is branching?

Think of a repository the central location for storing changes or commits.

- The `main` set of commits is like a trunk of a tree, where the commits are added in sequence (aka main).
- A `branch` is a fork in the tree, where we “split off” work and diverge from one of the commits.

Branches diverge from a specific commit, and do not include changes that happened on the trunk after the branch occurred.

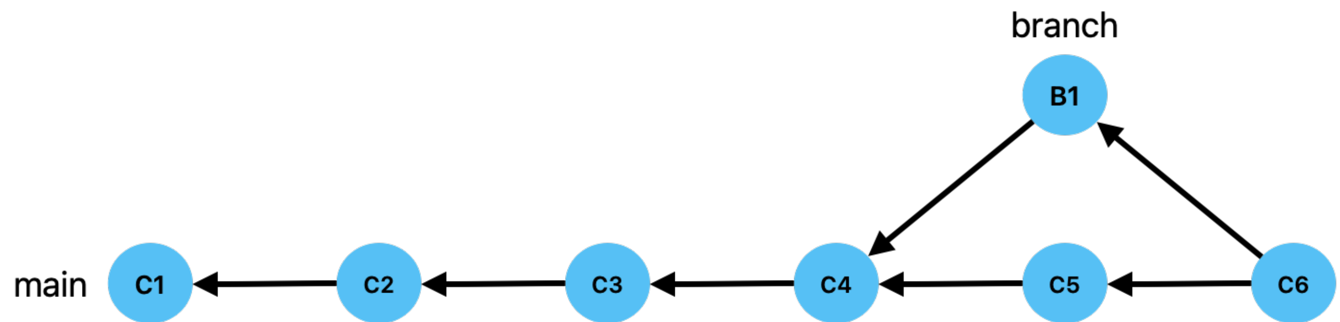

Figure 6: Commits and how they depend on one another. C1 is the first commit; the others build on that history. B1 diverges from `main`, but is eventually merged back into `main` in C6.

What are feature branches?

Branches are a mechanism in Git to reduce the likelihood of conflicting changes being made.

- We create a branch is to isolate our work from other changes on the trunk.
- Once we have a feature implemented and tested, we can merge our changes back into the trunk (“integration”).

Branches created for this purpose are called **feature branches** and are meant to isolate untested work.

A feature-branch workflow would be:

1. Create a feature branch from `main` for a feature that you are developing.
2. Make your changes on this branch. Test everything.
3. (Optional) Have changes code reviewed by someone on your team.
4. When done, merge from your feature branch back to the `main` branch.
5. (Optional) Delete the feature branch once the merge is completed.

What are feature branches?

```
$ git checkout -b test // create branch
Switched to a new branch 'test'

$ vim file1.md // make some changes
$ git add file1.md
$ git commit -m "Committing changed to file1.md"

$ git checkout master // switch to master
$ git merge test // merge changes from test
Updating 09e1947..ebb5838
Fast-forward
file1.md | 136 +++++++++++++++++++++++++++++++++++++++++
1 file changed, 118 insertions(+), 18 deletions(-)

$ git branch -d test // remove branch
Deleted branch test (was ebb5838).
```

Branching Models

Trunk-Development

In trunk-based development, all developers create feature branches from `main`, and merge changes directly back to main (after testing).

Characteristics

- Feature branches are short-lived.
- Development is continuous. Merges are frequent and issues are easier to resolve.
- There is no release branch. Releases are cut directly from main, and tagged by version.
- Integration testing becomes critical!

Pro/Con

- Continuous development and integration makes identifying and fixing issues easier.
- Limited ability to manage releases prior to the current release.

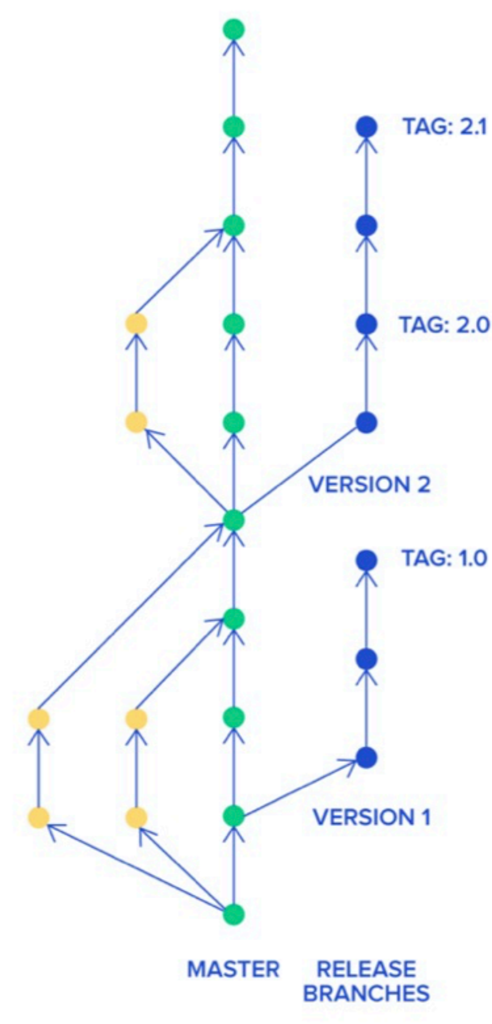

Figure 7: Trunk-Based Development

Trunk-Development

What does this look like in practice?

1. Create feature branches for development. Everyone does this.
2. All work is done on feature branches. You can commit, write tests etc. and iterate on the branch until that feature is complete.
2. When it's done, merge changes from the feature branch to `main`.
   - Each person typically does this for their own features.
   - Focus on getting a single feature complete and merged, vs working on many features simultaneously.
   - ONLY commit to `main` once the merge is complete, warnings and errors are resolved and all tests pass.
3. Create any software releases from the `main` branch.

Tip

Run tests in the feature branch prior to merging, and again on main before committing the merged code.

Git Flow Development

Git Flow is a more advanced model that is used when you need to maintain multiple versions of your software.

In this model, you have two main branches:

- `Develop` where all development takes place.
- `Main` which is only used for releases.

Characteristics

- Feature branches are created on `Develop`, and merged back to that branch when complete.
- Eventually Develop is merged back to Main. Product release branches are created from Main.

Pro/Con

- Long-lived feature branches; merges are difficult.
- Releases require an extra step, but are isolated.

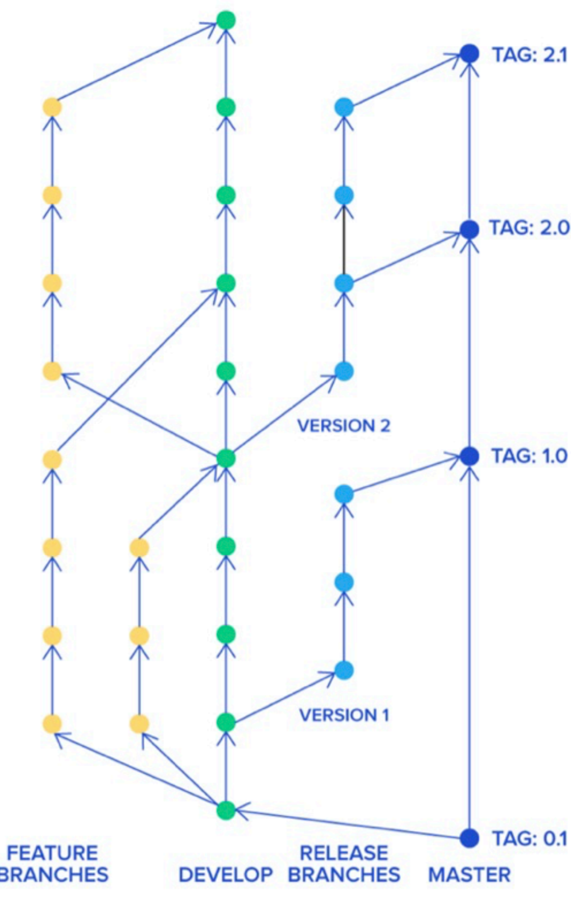

Figure 8: Git-Flow Development

Bibliography

[1] S. Chacon and B. Straub, *Git Pro*. Apress, 2014.