

# Introduction to Version Control Systems 1

DATA 4010 Seminar – Fall 2023

Stefano Ansaloni

University of Manitoba

September 18, 2023



**University  
of Manitoba**



**Digital Research  
Alliance** of Canada

# Stefano Ansaloni

Cloud Computing Specialist at University of Manitoba  
(part of the HPC support team)

Software Developer and DevOps Specialist since 2017

Linux User/Admin since 2005

# What is a version control system?

From Wikipedia ([Version Control](#)):

*Version control (also known as revision control, source control, or source code management) is a class of systems responsible for managing changes to computer programs, documents, large web sites, or other collections of information.*

## Revision Control System (RCS)

- ▶ First release in 1982
- ▶ Latest stable release in 2022
- ▶ Operates only on single files
- ▶ Only one user can work on a file at a time
- ▶ No network support

## Concurrent Versions System (CVS)

- ▶ First release in 1990
- ▶ Latest stable release in 2008
- ▶ Based on RCS (front-end to RCS)
- ▶ Repository-level change tracking
- ▶ Client-server model

## Subversion (SVN)

- ▶ First release in 2000
- ▶ Latest stable release in 2022
- ▶ Commits as true atomic operations
- ▶ Repository-level change tracking
- ▶ Path-based authorization

# History

## Git

- ▶ First release in 2005
- ▶ Latest stable release in 2023
- ▶ Distributed approach
- ▶ Non-linear workflows
- ▶ Safeguards against corruption (accidental or malicious)

# Distributed VS Centralized

| <b>Distributed VCS</b>                                      | <b>Centralized VCS</b>                      |
|---|---|
| peer-to-peer approach                                       | client-server approach                      |
| users can work offline                                      | users need access to the central server     |
| local copies can function as remote backups                 | central server is a single point of failure |
| complete codebase (including full history) mirrored locally | only requested revision mirrored locally    |
| slower initial checkout                                     | faster initial checkout                     |
| more storage required                                       | less storage required                       |





# Workflow examples

## Centralized Workflow

One central hub (or repository) can accept code, and everyone synchronizes their work with it.

The repository is usually accessed through a network connection to ease the code sharing among the developers.

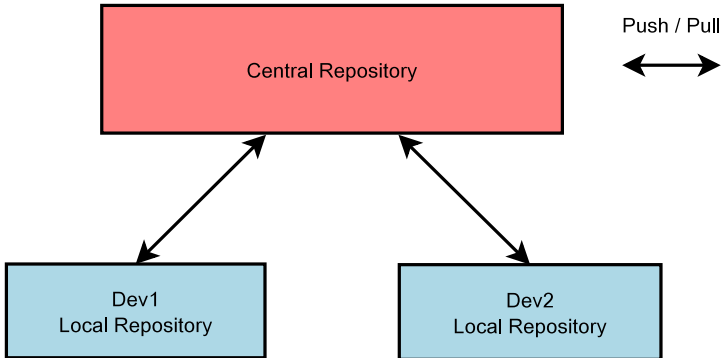
If two developers clone from the hub and both make changes, the first to push the changes to the repository can do so with no problems.

The second developer must merge in the first one's work before pushing the local changes.



# Workflow examples

## Centralized Workflow



# Workflow examples

## Integration-Manager Workflow

The project maintainer creates the official repository.

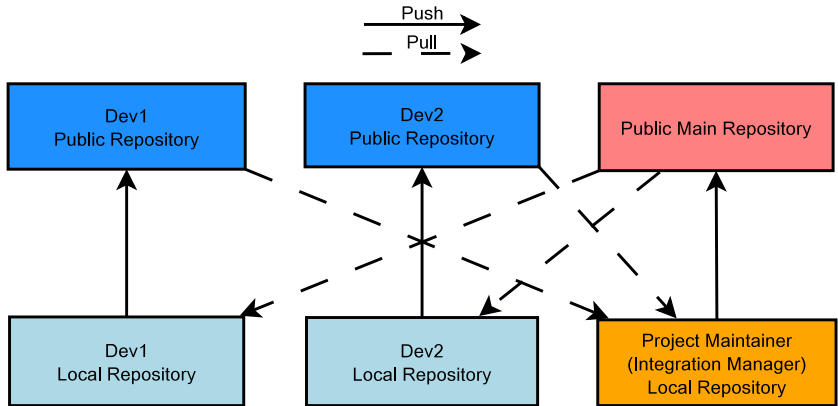
Developers create public clones of the official repository, and start adding changes.

When a developer is done, sends a pull-request to the project maintainer.

The maintainer reviews the changes and decides whether to merge them or not into the official repository.

# Workflow examples

## Integration-Manager Workflow



## Dictator and Lieutenants Workflow

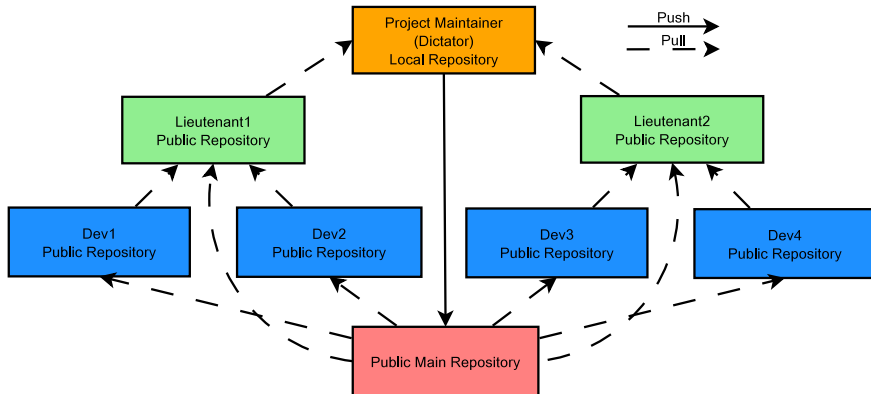
This is a variant of the Integration-Manager workflow, where a second level of “integration-managers” is added.

This workflow is usually adopted for huge projects with hundreds (or thousands) of collaborators.



# Workflow examples

## Dictator and Lieutenants Workflow



# What is a repository?

A *repository* is a data structure that stores metadata for a set of files or directory structure.

The main purpose of a *repository* is to store information about a set of files, as well as the history of changes made to those files.

In the case of Git, the whole set of information is duplicated on every user's system.

# What is a working copy?

The *working copy* (or *working tree*) is the local copy of files from a repository, at a specific time or revision.

All work done to the files in a repository is initially done on a *working copy*, and for this reason it could be seen as a sandbox.



**University  
of Manitoba**



# What is a commit?

A *commit* (or *revision*) is a set of alterations packaged together, along with meta information about those alterations.

It describes the exact differences between two successive versions in the version control system's repository of changes.

*Commits* are typically treated as an atomic unit.

# What is a commit operation?

A *commit operation* is an action which saves the changes made on the working copy to the repository, creating a new revision of the repository.

# What is a tag?

A *tag* is a textual label that can be associated with a specific commit.

This allows to define a meaningful name to be given to a particular state of the project.

# What is a branch?

A *branch* is a duplicated set of files that allows the two copies of those files to be independently developed at different speeds, or in different ways.

# What is a merge operation?

A *merge operation* (or *merging*) is the action of reconciling multiple changes made to a set of files.

Usually, this is necessary when one or more files are modified on two independent branches.

The result is a single collection of files that contains both sets of changes.



University  
of Manitoba

## Git – What is a remote?

A *remote* (or *remote repository*) is a copy of a local repository hosted on the Internet or network somewhere.

*Remote repositories* are used to ease the collaboration between multiple developers by pushing and pulling data to and from them when they need to share work.



University  
of Manitoba

## Git – What is the staging area?

The *staging area* is a place to record files before committing them.

All (and only) files inside the *staging area* will be taken into account when creating a commit.



**University  
of Manitoba**

# Git – What is a branch? (again)

Two possible definitions:

1. a name for a particular commit and all the commits that are ancestors of it
2. a line of development (or better: a directed acyclic graph (DAG) of development, where the commits represent the graph's nodes)

A *branch* can be local (if it exists in a local repository), remote (if it exists on a configured remote), or both.



## Git – What is a tag? (again)

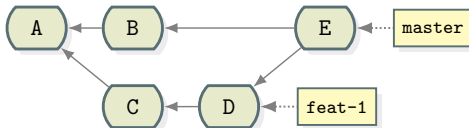
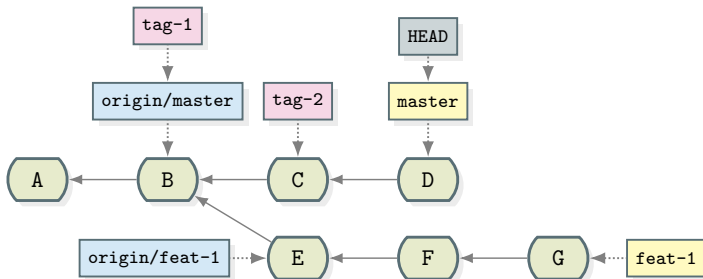
A *tag* is a reference that points to a specific commit.

The pointed commit never changes (no further history of commits).

A *tag* is like a branch that does not change.



# Git branch examples



# Creating a Git repository

To create a new local Git repository, move to the desired directory and execute “`git init`” (this will create an empty Git repository in the current directory).

If you want to create a local copy of a remote repository, you can use “`git clone <remote_url> [<local_dir>]`”.



# Checking the status of a Git repository

To determine which files are in which state within a Git repository, you can use “git status”.

---

```
user@host:~/repo$ git status
On branch master
```

```
No commits yet
```

```
nothing to commit (create/copy files and use
"git add" to track)
```

---



**University  
of Manitoba**

# Adding files to a Git repository

By default Git does not keep track of newly added files.

---

```
user@host:~/repo$ touch file1
user@host:~/repo$ git status
On branch master
```

No commits yet

Untracked files:

```
(use "git add <file>..." to include in what will be
    committed)
file1
```

```
nothing added to commit but untracked files present
    (use "git add" to track)
```

---



**University  
of Manitoba**

# Adding files to a Git repository

To add a new file to a Git repository you can use  
“git add <filename\_or\_directory>”.

---

```
user@host:~/repo$ git add file1
user@host:~/repo$ git status
On branch master
```

No commits yet

Changes to be committed:  
 (use "git rm --cached <file>..." to unstage)  
 new file: file1

---



**University  
of Manitoba**

# Adding commits to a Git repository

To add a new commit to a Git repository, the staging area must not be empty, then you can use “git commit” to commit the changes.

---

```
user@host:~/repo$ git commit -m "Initial commit"
[master (root-commit) 99fca70] First commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1
user@host:~/repo$ git status
On branch master
nothing to commit, working tree clean
```

---



**University  
of Manitoba**

# Viewing commits history of a Git repository

To show the history of a repository, you can use “git log”.

---

```
user@host:~/repo$ git log
commit 99fca709626fc96ac1c2f744f5f3b25feaf542c6 (HEAD
-> master)
Author: Name Here <email_here@example.org>
Date:    Sun Jan 1 00:00:00 2023 -0500

    First commit
```

---



## Unstaging files in a Git repository

If you mistakenly added a file to the staging area, you can remove it using “`git reset <filename>`”, or “`git restore --staged <filename>`”.

---

```
user@host:~/repo$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   file1
    new file:   file2
user@host:~/repo$ git reset file1
Unstaged changes after reset:
M file1
```

---



## Restoring files in a Git repository

To restore a file to a previous commit, you can use  
“git checkout <ref> -- <filename>”, or  
“git restore -s <ref> <filename>”.

---

```
user@host:~/repo$ git log --oneline
546bb0c (HEAD -> master) Second commit
99fca70 First commit
user@host:~/repo$ git checkout 99fca70 -- file1
user@host:~/repo$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   file1
user@host:~/repo$ git restore --staged file1
user@host:~/repo$ git restore file1
```

---



University  
of Manitoba

# Managing branches in a Git repository

To list, create, or delete branches you can use “git branch”.

To switch branch you can use “git checkout <branch\_name>”, or “git switch <branch\_name>”.

---

```
user@host:~/repo$ git branch
* master
user@host:~/repo$ git branch new-branch
user@host:~/repo$ git branch
* master
  new-branch
user@host:~/repo$ git switch new-branch
Switched to branch 'new-branch'
user@host:~/repo$ git branch
  master
* new-branch
user@host:~/repo$ git switch master
user@host:~/repo$ git branch -d new-branch
Deleted branch new-branch (was 99fca70).
```



**University  
of Manitoba**

# Merging branches in a Git repository

To merge two branches, you can use  
“git merge <src\_branch\_name>”.

---

```
user@host:~/repo$ git switch -c fix
Switched to a new branch 'fix'
user@host:~/repo$ touch file2
user@host:~/repo$ git add file2
user@host:~/repo$ git commit -am "Important fix"
[fix c87f330] Important fix
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file2
user@host:~/repo$ git switch master
Switched to branch 'master'
user@host:~/repo$ git merge fix
Updating 546bb0c..c87f330
Fast-forward
 file2 | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file2
```



**University  
of Manitoba**

# Managing remotes in a Git repository

To list, create, or delete remotes you can use “git remote”.

Usually Git commands use “*origin*” as default remote name when no specific configuration is present.

---

```
user@host:~/repo$ git remote
user@host:~/repo$ git remote add src-git https://
    github.com/git/git
user@host:~/repo$ git remote
src-git
user@host:~/repo$ git fetch src-git
...
user@host:~/repo$ git branch -r
    src-git/master
    src-git/next
    src-git/todo
...
user@host:~/repo$ git remote remove src-git
```

---



University  
of Manitoba

# Synchronizing changes from/to a remote in a Git repository

To download changes from a remote, you can use

“git fetch <remote\_name>” (a “git merge” is needed to update the local branch).

To upload changes to a remote, you can use

“git push <remote\_name> <branch\_name>”.

---

```
user@host:~/repo$ git fetch src-git
remote: Enumerating objects: 354353, done.
remote: Counting objects: 100% (913/913), done.
remote: Compressing objects: 100% (913/913), done.
remote: Total 354347 (delta 0)                <...>
Receiving objects: 100% (354347/354347)    <...>
Resolving deltas: 100% (266375/266375), done.
From https://github.com/git/git
* [new branch]      master      -> src-git/master
* [new branch]      next        -> src-git/next
* [new branch]      todo        -> src-git/todo
* [new tag]         v2.42.0     -> v2.42.0
...
```



# Synchronizing changes from/to a remote in a Git repository

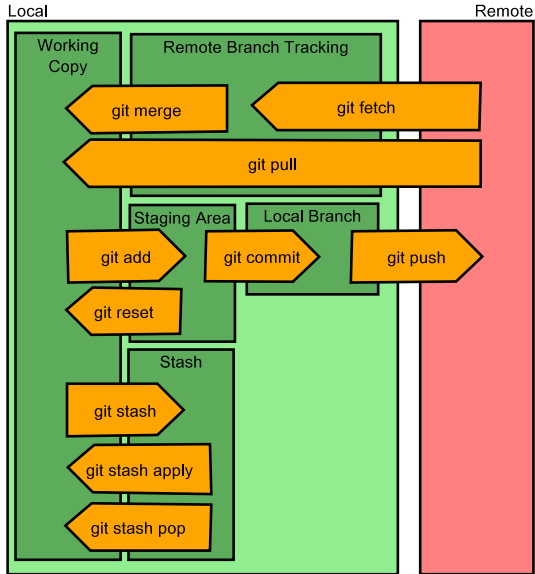
If the current branch is set up to track a remote branch, you can use “`git pull`” to automatically fetch and then merge that remote branch into the current branch.

Using “`git pull`” generally fetches data from the originally cloned server and automatically tries to merge it into the code you’re currently working on.



**University  
of Manitoba**

# Git cheat-sheet





# Git GUI clients

Free, open-source, multiplatform (Windows, Mac, Linux):

- ▶ MeGit – <https://github.com/eclipsesource/megit>
- ▶ Gitnuro – <https://github.com/JetpackDuba/Gitnuro>
- ▶ Gittyup – <https://github.com/Murmele/Gittyup>

Free, closed-source, multiplatform (Windows, Mac):

- ▶ Sourcetree – <https://www.sourcetreeapp.com/>

Note: remember that the only real client is the "git" command line tool.



# Useful links

- ▶ Official Git documentation: <https://git-scm.com/doc>
- ▶ “Pro Git” book (free): <https://git-scm.com/book>

THIS IS GIT. IT TRACKS COLLABORATIVE WORK  
ON PROJECTS THROUGH A BEAUTIFUL  
DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL  
COMMANDS AND TYPE THEM TO SYNC UP.  
IF YOU GET ERRORS, SAVE YOUR WORK  
ELSEWHERE, DELETE THE PROJECT,  
AND DOWNLOAD A FRESH COPY.



# Questions?

Thank you