

Data management on HPC platforms

Transferring data and handling source code with Git

The SCITAS team

<https://scitas-doc.epfl.ch>

23rd of May 2024

- Table of Contents
- Data management at SCITAS
 - ▶ SCITAS' file systems
 - ▶ Storing your data
 - ▶ Object storage using S3
- Transferring data
 - ▶ Transferring data
 - ▶ Transferring data to/from the outside of EPFL network
- Handling source code with Git
 - ▶ Resources
 - ▶ What is version control?
 - ▶ A brief history of Git
 - ▶ Git internals
 - ▶ Using Git locally to save you work
 - Getting a repository

- Getting a repository
- Checking the status
- Tracking your changes
- Committing your changes
- Ignoring files
- Viewing the history
- Viewing the differences
- Tagging
- Working with branches
- ▶ Interacting with the world!
 - Showing the remotes
 - Adding a remote
 - Getting changes from the remotes
 - Pushing your work to the remotes
- ▶ Configuring Git



Data management at SCITAS



- There are seven different file systems on the clusters.
- Each of them has a purpose for which it is best suited for.
- But also particularities that could lead to data loss if not used correctly.
- It is then very important to know the characteristics of each in order to efficiently use the clusters.

- /home is the default file system on which you are when you log into the machines.
- It is used to store relatively small files that you want to conserve, e.g. your source codes, configuration files, and input files.

- /home is the default file system on which you are when you log into the machines.
- It is used to store relatively small files that you want to conserve, e.g. your source codes, configuration files, and input files.
- There is a quota of 100 GB (use the `fsu` command to check it).
 - ▶ Note that due to data replication, `fsu` will print twice your usage.
- It is accessible from all the clusters.

- /home is the default file system on which you are when you log into the machines.
- It is used to store relatively small files that you want to conserve, e.g. your source codes, configuration files, and input files.
- There is a quota of 100 GB (use the `fsu` command to check it).
 - ▶ Note that due to data replication, `fsu` will print twice your usage.
- It is accessible from all the clusters.
- Data is backed up
- Data will be erased after two years of inactivity or six months after leaving EPFL.

- /work is a team storage that is accessible by all your colleagues.
- It is ideal to store large data such as your simulation results for several years.

- /work is a team storage that is accessible by all your colleagues.
- It is ideal to store large data such as your simulation results for several years.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.

- /work is a team storage that is accessible by all your colleagues.
- It is ideal to store large data such as your simulation results for several years.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.
- Data is not backed up by default, but you can activate it by creating a file called `.backup` at the root of your /work, e.g.

```
> touch /work/scitas/.backup
```
- Data will be erased six months after cessation of payment.

- /archive is a particular file system that is stored on tape.
- It is used to store data for very-long time.

- /archive is a particular file system that is stored on tape.
- It is used to store data for very-long time.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.

- /archive is a particular file system that is stored on tape.
- It is used to store data for very-long time.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.
- There is no backup, but data is replicated on two different sites.
- Data will be erased six months after cessation of payment.

- /archive is a particular file system that is stored on tape.
- It is used to store data for very-long time.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.
- There is no backup, but data is replicated on two different sites.
- Data will be erased six months after cessation of payment.
- Plan your data archiving. The EPFL library can assist you:
<https://www.epfl.ch/campus/library/services-researchers/#rdm>

- /archive is a particular file system that is stored on tape.
- It is used to store data for very-long time.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.
- There is no backup, but data is replicated on two different sites.
- Data will be erased six months after cessation of payment.
- Plan your data archiving. The EPFL library can assist you:
<https://www.epfl.ch/campus/library/services-researchers/#rdm>

- Used for cold storage and not frequent IO. There is a limited bandwidth (a robotic arm has to physically move tapes around).
- Make sure to follow the good practices
<https://scitas-doc.epfl.ch/user-guide/data-management/archiving/>

- /transfer is used to transfer your data outside of the EPFL network.

- /transfer is used to transfer your data outside of the EPFL network.
- There is no quota.
- It is accessible from all the clusters and the `scitas-transfer.epfl.ch` node.

- /transfer is used to transfer your data outside of the EPFL network.
- There is no quota.
- It is accessible from all the clusters and the `scitas-transfer.epfl.ch` node.
- There is no backup.
- Data will be erased after 15 days.

- /transfer is used to transfer your data outside of the EPFL network.
 - There is no quota.
 - It is accessible from all the clusters and the `scitas-transfer.epfl.ch` node.
 - There is no backup.
 - Data will be erased after 15 days.
-
- We will see how it works in the next section.

- /export provides a storage space where your data can be exported to other machines using the NFS or SMB.

- /export provides a storage space where your data can be exported to other machines using the NFS or SMB.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.

- /export provides a storage space where your data can be exported to other machines using the NFS or SMB.
- There is no quota. Storage is based on a pay-per-use model.
- It is accessible from all the clusters.
- Data is not backed up by default, but you can activate it by creating a file called `.backup` at the root of your `/export`, e.g.
> `touch /export/scitas/.backup`
- Data will be erased six months after cessation of payment.

- /scratch is a temporary high-performance storage for your simulation data.

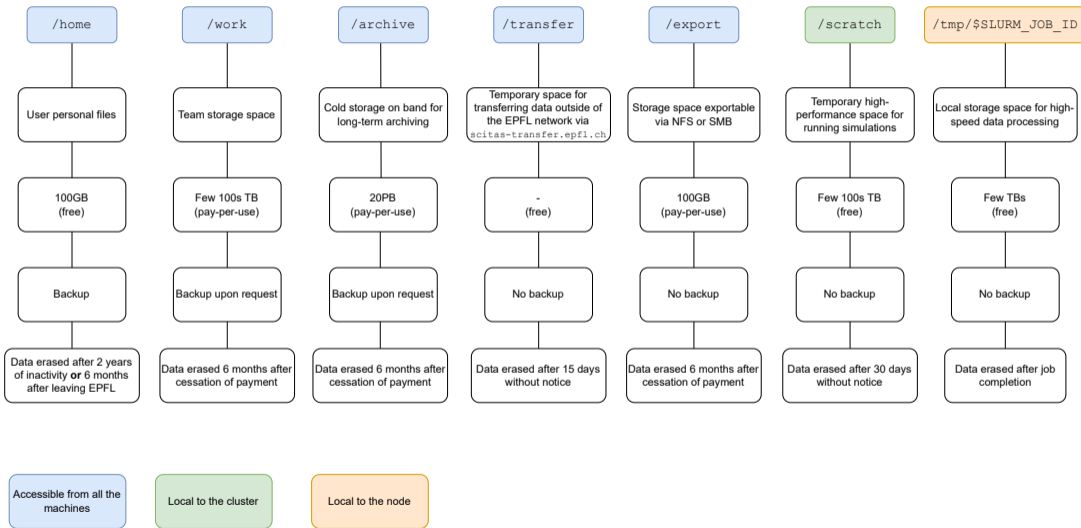
- `/scratch` is a temporary high-performance storage for your simulation data.
- There is no quota.
- Each cluster has its own `/scratch`.

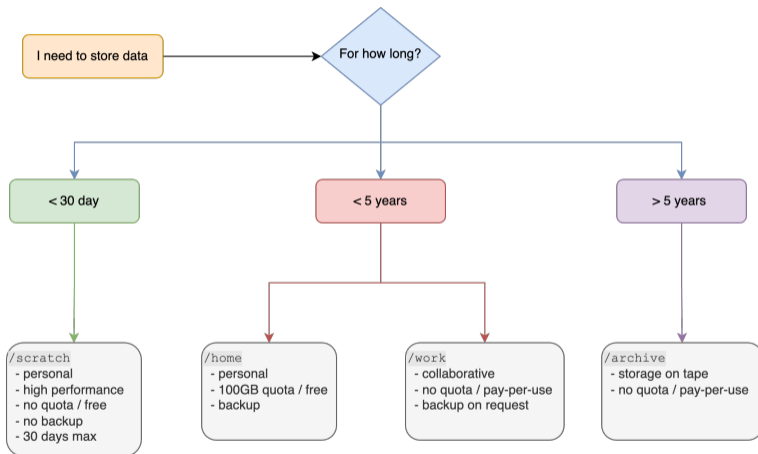
- /scratch is a temporary high-performance storage for your simulation data.
- There is no quota.
- Each cluster has its own /scratch.
- There is no backup.
- Files older than 30 days are erased without notice every day.
- There may be file deletion (<30 days) at any time to preserve system integrity.

- /tmp/\$SLURM_JOB_ID is a node-local and temporary high-performance storage for your simulation data.

- /tmp/\$SLURM_JOB_ID is a node-local and temporary high-performance storage for your simulation data.
- There is no quota.
- Each node has its own /tmp.

- `/tmp/$SLURM_JOB_ID` is a node-local and temporary high-performance storage for your simulation data.
- There is no quota.
- Each node has its own `/tmp`.
- There is no backup.
- Files are automatically removed after the run terminates.





- SCITAS supports object storage using S3.
- Data is stored on the central storage, *but* there is no backup nor replication.
- IBM implementation so limited features compared to Amazon.
- Prices are the same as work.

- SCITAS supports object storage using S3.
- Data is stored on the central storage, *but* there is no backup nor replication.
- IBM implementation so limited features compared to Amazon.
- Prices are the same as work.
- To create an account:
 - ▶ Send an email to to 1234@epfl.ch with “SCITAS S3” as the subject.
 - ▶ We will create an s3_cred.txt file in your /home containing you credentials (please store them safely and remove the file).

- We do not provide a web interface to manage your buckets (like Amazon)
- We support three tools:
 - ▶ Rclone
 - ▶ S3cmd
 - ▶ Cyberduck
- For this tutorial we will only use Rclone.

■ Using to documentation:

- ▶ Configure rclone
 - The credentials can be found in `/work/scitas-share/data_management_course`
- ▶ Create a bucket
- ▶ List the buckets
- ▶ Upload a single file and a folder
- ▶ Play with it!



Transferring data



There are different protocols that you can use to transfer data between two machines:

- `scp` Copy files between a local and a remote host.
- `rsync` Synchronize files between two locations.
- `sftp` Protocol allowing the user file access, copy and management over the network.
- `sshfs` Mount a remote file system to use it as if it were local.

They are all based of the SSH protocol.

- `rsync` is a tool used to synchronize files.
- It compares the modification times and sizes of files to transfer only the differences.
- If the transfer is interrupted, can be resumed from where it stopped.

- `rsync` is a tool used to synchronize files.
- It compares the modification times and sizes of files to transfer only the differences.
- If the transfer is interrupted, can be resumed from where it stopped.
- Usage:

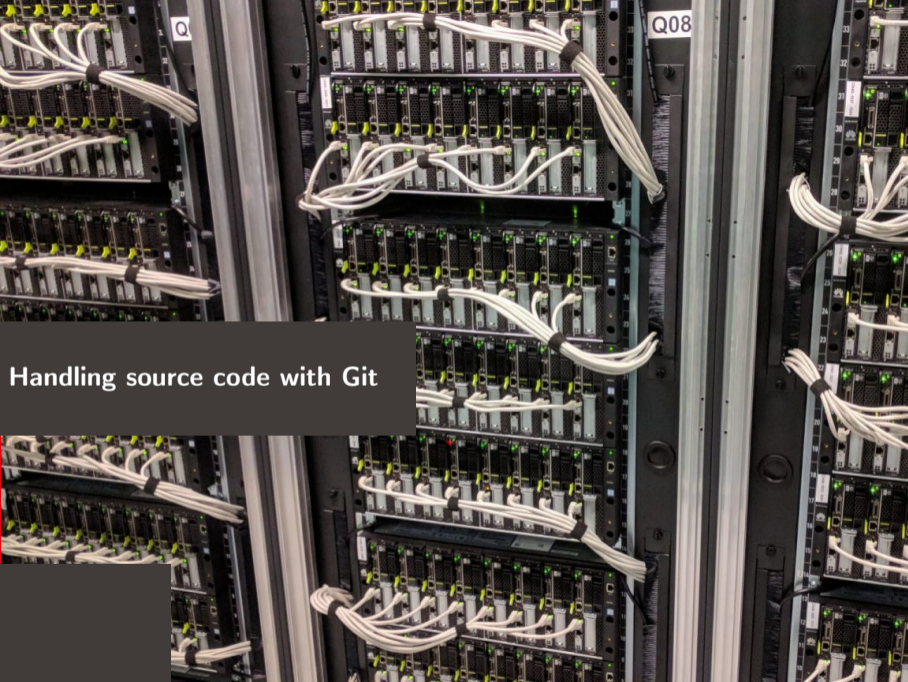
- ▶ `rsync -avP <source> <destination>`
- ▶ `rsync -avP <user>@<remote>:<source> <destination>`
- ▶ `rsync -avP <source> <user>@<remote>:<destination>`

where the options are

- ▶ `-a` for a recursive copy preserving modification times, links, permissions, etc.
- ▶ `-u` to preserve files that are newer on the destination,
- ▶ `-v` to have a verbose output,
- ▶ `-P` to keep partial files and print the progress.

- Create a temporary folder, `tmp/`, in your `/home`.
- Copy the folder `/work/scitas-share/cmake-3` in your `tmp/` folder:
`cp -r /work/scitas-share/cmake-3 tmp/`
- Create a backup folder, `backup/`, in your `tmp/` folder.
- Use `rsync` to create a backup of `tmp/cmake-3` into `tmp/backup`.
- Add a random modification in `tmp/cmake-3/doc/cmake-3.11/Copyright.txt`
- Re-synchronize `tmp/cmake-3` into `tmp/backup`.
- Transfer the backup folder to your local machine.
- What happens if you add a trailing slash to the source directory, *i.e.* the difference between the following two
`rsync -auvP source destination`
and
`rsync -auvP source/ destination`

- You may sometimes need to transfer data between the SCITAS' clusters and other machines outside of EPFL network, e.g. another HPC center.
- To do so, we have a transfer node, `scitas-transfer.epfl.ch`, with limited capabilities "accessible" from outside of EPFL network.
 - ▶ For external access, you must use `rclone`.
 - ▶ If it is not possible, we can open an IP address upon request.
- For security reasons, only the `/transfer` file system is mounted on `scitas-transfer`.
- To transfer your data from EPFL to another center, you typically need to do the following:
 - ▶ From a cluster, copy the data you wish to transfer into `/transfer`.
 - ▶ Log into the destination machine and transfer the files using your preferred protocol, e.g.
`rsync -auvP <user>@scitas-transfer.epfl.ch:/transfer/<folder> <local folder>`



Handling source code with Git



- Git official documentation:
<https://git-scm.com/doc>
- Pro Git, Scott Chacon and Ben Straub:
<https://git-scm.com/book/en/v2>
- Many images are taken from this book.
- A funny game to learn Git interactively:
<https://learngitbranching.js.org/>

- Version control is the act of recording the state of files over time.

- Version control is the act of recording the state of files over time.
 - ▶ You have certainly all used this archaic version control



- Version control is the act of recording the state of files over time.

- ▶ You have certainly all used this archaic version control



- Version control system (VCS) is a software that automates version control.
- In particular, it allows you to:
 - ▶ revert to a specific state,
 - ▶ compare two states,
 - ▶ see who last modified a file,
 - ▶ etc.

- Version control is the act of recording the state of files over time.

- ▶ You have certainly all used this archaic version control



- Version control system (VCS) is a software that automates version control.

- In particular, it allows you to:

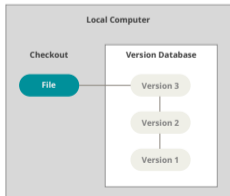
- ▶ revert to a specific state,
 - ▶ compare two states,
 - ▶ see who last modified a file,
 - ▶ etc.

- These capabilities are very handy because:

- ▶ Bugs will be introduced,
 - ▶ You want to test different developments,
 - ▶ Sometimes you need a timestamp on some code, e.g. for patent issues.

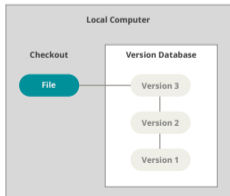
Local VCS

- A unique repository is created on your computer.
- It contains the files as well as a “version database”.
- Contribution from one user.
- For example, RCS.



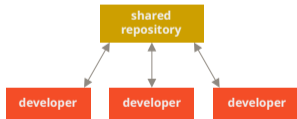
Local VCS

- A unique repository is created on your computer.
- It contains the files as well as a “version database”.
- Contribution from one user.
- For example, RCS.



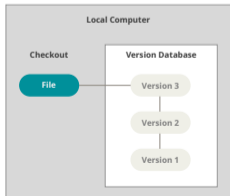
Centralized VCS

- There is a single shared repository.
- It's the single source of truth.
- Everyone pulls/pushes information from/to it.
- For example, CVS and Subversion.



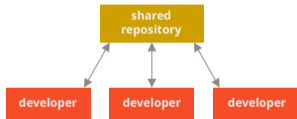
Local VCS

- A unique repository is created on your computer.
- It contains the files as well as a “version database”.
- Contribution from one user.
- For example, RCS.



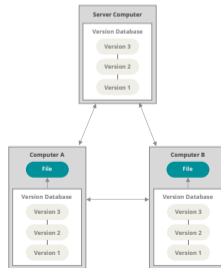
Centralized VCS

- There is a single shared repository.
- It's the single source of truth.
- Everyone pulls/pushes information from/to it.
- For example, CVS and Subversion.



Distributed VCS

- Everyone gets the whole repo.
- No single source of truth anymore.
- Can pull from one and push to the other.
- For example Git and Mercurial.



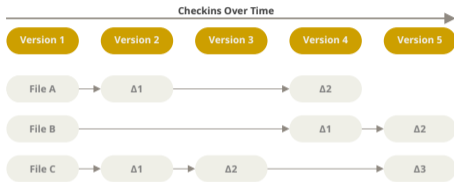
- Git development started in 2005 by Linus Torvalds.
- The goal was to replace BitKeeper for the Linux kernel development.
- The design goals were:
 - ▶ Speed
 - ▶ Simple design
 - ▶ Strong support for non-linear development
 - ▶ Fully distributed
 - ▶ Able to handle large projects efficiently



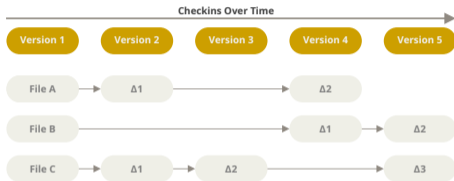
- Git development started in 2005 by Linus Torvalds.
- The goal was to replace BitKeeper for the Linux kernel development.
- The design goals were:
 - ▶ Speed
 - ▶ Simple design
 - ▶ Strong support for non-linear development
 - ▶ Fully distributed
 - ▶ Able to handle large projects efficiently



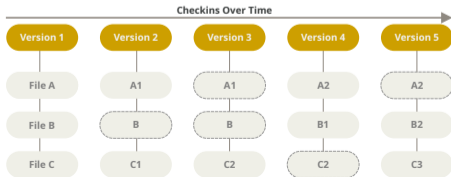
- Most of the VCSs store information as a list of file-based changes:



- Most of the VCSs store information as a list of file-based changes:



- Git, on the other hand, stores them as whole-file snapshots:

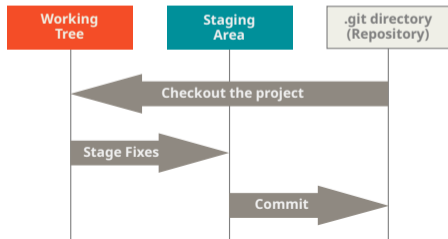


- Almost every operation in Git is local
 - ▶ No latency due to network
 - ▶ Can work even without an internet connection

- Almost every operation in Git is local
 - ▶ No latency due to network
 - ▶ Can work even without an internet connection
- Git has integrity
 - ▶ Everything in Git is checksummed before being stored
 - ▶ Impossible to change the content of any file without Git knowing it
 - ▶ Git uses SHA-1 hashes (40-character string):
24b9da6552252987aa493b52f8696cd6d3b00373

- Almost every operation in Git is local
 - ▶ No latency due to network
 - ▶ Can work even without an internet connection
- Git has integrity
 - ▶ Everything in Git is checksummed before being stored
 - ▶ Impossible to change the content of any file without Git knowing it
 - ▶ Git uses SHA-1 hashes (40-character string):
24b9da6552252987aa493b52f8696cd6d3b00373
- Git usually only adds data to the database. It is difficult to lose information once committed.

- Git has four main states in which a file can reside in, *untracked*, *modified*, *staged*, *committed*:
 - ▶ Untracked means that the file is not tracked by Git.
 - ▶ Modified means that you have changed the file, but not saved it to the database yet.
 - ▶ Staged means that the file is marked to be included in the next commit.
 - ▶ Committed means that the data is safely stored into the local database.



With Git, there are two ways of getting a repository:

- Initialize an empty repository:

```
$ git init
Initialized empty Git repository in /home/user/my_code/.git/
```

- Get an existing repository:

```
$ git clone <repository address> Cloning into 'my_code'... remote:
Counting objects: 30989, done. remote: Compressing objects:
100% (7773/7773), done.
remote: Total 30989 (delta 23002), reused 30984 (delta 22999)
Receiving objects:
100% (30989/30989), 74.67 MiB | 14.82 MiB/s, done.
Resolving deltas: 100% (23002/23002), done.
```



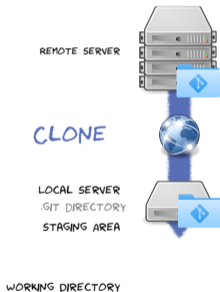
With Git, there are two ways of getting a repository:

- Initialize an empty repository:

```
$ git init
Initialized empty Git repository in /home/user/my_code/.git/
```

- Get an existing repository:

```
$ git clone <repository address> Cloning into 'my_code'... remote:
Counting objects: 30989, done. remote: Compressing objects:
100% (7773/7773), done.
remote: Total 30989 (delta 23002), reused 30984 (delta 22999)
Receiving objects:
100% (30989/30989), 74.67 MiB | 14.82 MiB/s, done.
Resolving deltas: 100% (23002/23002), done.
```



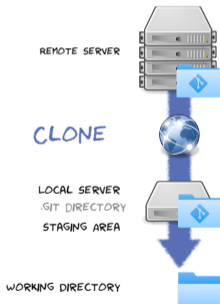
With Git, there are two ways of getting a repository:

- Initialize an empty repository:

```
$ git init
Initialized empty Git repository in /home/user/my_code/.git/
```

- Get an existing repository:

```
$ git clone <repository address> Cloning into 'my_code'... remote:
Counting objects: 30989, done. remote: Compressing objects:
100% (7773/7773), done.
remote: Total 30989 (delta 23002), reused 30984 (delta 22999)
Receiving objects:
100% (30989/30989), 74.67 MiB | 14.82 MiB/s, done.
Resolving deltas: 100% (23002/23002), done.
```



- Make sure you have Git installed by typing

```
$ git --version  
git version 2.39.2
```

- Clone the following repo:
`https://gitlab.epfl.ch/lanti/data-management-exercises.git`
- Go into the exercise folder and initialize it:
`cd data-management-exercises`
`./setup.sh`

```
git status
```

To check the status of your repository, use `git status`.

To check the status of your repository, use `git status`.

```
$ git status
On branch main
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
modified:   file_1.txt

Changes not staged for commit:
(use "git add/rm <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
modified:   file_1.txt
modified:   file_2.txt
deleted:    file_4.txt

Untracked files:
(use "git add <file>..." to include in what will be committed)
file_5.txt
```

To add a file to the staging area, use `git add <file name>`

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   file_1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

To add a file to the staging area, use `git add <file name>`

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   file_1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git add file_1.txt
```

To add a file to the staging area, use `git add <file name>`

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   file_1.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
$ git add file_1.txt
```

```
$ git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
modified:   file_1.txt
```

Now that we have staged some files, we can save our work using `git commit`

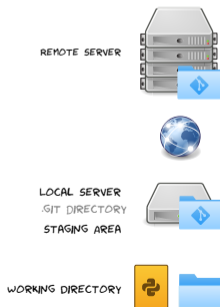
```
$ git commit -m <commit message>
   [main 52aafd1] <commit message>
1 file changed, 1 insertion(+)
```

If you omit the `-m` option, it will open the default editor to input your message.

Now that we have staged some files, we can save our work using `git commit`

```
$ git commit -m <commit message>
   [main 52aafd1] <commit message>
   1 file changed, 1 insertion(+)
```

If you omit the `-m` option, it will open the default editor to input your message.

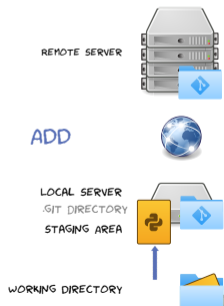


`git commit`

Now that we have staged some files, we can save our work using `git commit`

```
$ git commit -m <commit message>
   [main 52aafd1] <commit message>
   1 file changed, 1 insertion(+)
```

If you omit the `-m` option, it will open the default editor to input your message.



`git commit`

Now that we have staged some files, we can save our work using `git commit`

```
$ git commit -m <commit message>
[main 52aafd1] <commit message>
1 file changed, 1 insertion(+)
```

If you omit the `-m` option, it will open the default editor to input your message.



A few tips

- Commit related changes
- Commit often
- Don't commit half-done work
- Write good commit messages
 - ▶ Capitalized, short (50 chars or less) summary
 - ▶ Always leave the second line blank
 - ▶ More detailed explanatory text, if necessary
 - ▶ Write your commit message in the imperative: "Fix bug" and not "Fixed bug"

	COMMENT	DATE
○	CREATED MAIN LOOP & TIMING CONTROL	14 HOURS AGO
○	ENABLED CONFIG FILE PARSING	9 HOURS AGO
○	MISC BUGFIXES	5 HOURS AGO
○	CODE ADDITIONS/EDITS	4 HOURS AGO
○	MORE CODE	4 HOURS AGO
○	HERE HAVE CODE.	4 HOURS AGO
○	AAAAAAAAA	3 HOURS AGO
○	ADKFJBLKDFJSDKLFJ	3 HOURS AGO
○	MY HANDS ARE TYPING WORDS	2 HOURS AGO
○	HAHAHAHAANDS	2 HOURS AGO

AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

- There are files you will never want to include in the repository, e.g. *.o, CMake build directory, etc.
- To avoid mistakes, Git has a special file, .gitignore, to list excluded files

- There are files you will never want to include in the repository, e.g. *.o, CMake build directory, etc.
- To avoid mistakes, Git has a special file, .gitignore, to list excluded files

```
# ignore all .a files
*.a
# but do track lib.a, even though you're ignoring .a files above
!lib.a
# only ignore the TODO file in the current directory, not subdir/TODO
/TODO
# ignore all files in any directory named build
build/
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```

- One of the strength of Git is to be able to quickly inspect the whole history of your project.
- This is achieved using the `git log` command

```
commit 0d96e54298920a5d05abfecf62e002ad0b4281c2 (HEAD -> new_slides_2024)
Author: John Doe <john.doe@epfl.ch>
Date:   Mon May 13 10:43:43 2024 +0200

Add skeleton for new course

commit 53edfad5d2cc8ad1ffe9e9073ac71b4a88231f3e
Author: John Doe <john.doe@epfl.ch>
Date:   Mon May 13 09:06:40 2024 +0200

Remove previous content

commit bedb8f7c493ca82e365b8b7266454b93aa956422 (origin/main, origin/HEAD, main)
Author: Jane Doe <jane.doe@epfl.ch>
Date:   Wed Feb 16 11:46:56 2022 +0100

Minor updates for 2022 February course
```

- There are many options to `git log` that allow you to get various information.
- For example:
 - `--patch` : Show the changes made in the commits,
 - `--oneline` : Show a shortened version of the log,
 - `--graph` : Print the log in a graph form,
 - `--format` : Change the way (the format) the log is printed.

```
| * 29e379b - John Doe - Merge branch 'specific/bern'
| | \
| | * b4355e6 - Jane Doe - Use a shellscrip(7 years ago)
| | * 359aa4c - Jane Doe - Build the slides with docker(7 years ago)
| * | e86e31e - John Doe - Merge branch 'specific/bern'
| * | 5b2690c - John Doe - adding remote/conflicts(7 years ago)
| | /
| * 6498225 - John Doe - adding add/commit/push(7 years ago)
| * 52098e9 - Jane Doe - persons->people(7 years ago)
| /
* c4b2504 - John Doe - Update scratch(7 years ago)
```

- Use `git status` to see on which branch you are on. How is your working directory? Would you expect anything different?
- Create a file.
- What does `git status` output look like now?
- Add the file to the staging area. How does `git status` look now?
- Commit the file to the repository. How does `git status` look now?
- Inspect the history of the repository using `git log`:
 - ▶ Who made the first commit, and when?
 - ▶ What is the hash of the commit adding the `README.md` file?
 - ▶ Try different options, e.g. `--oneline`, `--graph`, and `--patch`.
- Change an existing file. How does `git status` look now?
- Commit the changes.
- Execute the following command:
`./utils/generate_files.sh`
- Configure Git to accept only PDF files that are in `images/cats` and subfolders.
- Commit those images.

- One of the very important feature of Git is to easily get the differences between two files.
- This is done using `git diff`.

```
$ git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
modified:   program.cpp

no changes added to commit (use "git add" and/or "git commit -a")
```

`git diff`

- One of the very important feature of Git is to easily get the differences between two files.
- This is done using `git diff`.

```
$ git diff
diff --git a/program.cpp b/program.cpp
index f37606f..76edb6c 100644
--- a/program.cpp
+++ b/program.cpp
@@ -2,7 +2,7 @@

int main(int argc, char* argv[]) {

- if (argc = 1) {
+ if (argc == 1) {
     std::cout << "Not enough arguments\n";
 }
}
```

`git diff`

- One of the very important feature of Git is to easily get the differences between two files.
- This is done using `git diff`.

```
$ git diff
diff --git a/program.cpp b/program.cpp
index f37606f..76edb6c 100644
--- a/program.cpp
+++ b/program.cpp
@@ -2,7 +2,7 @@

int main(int argc, char* argv[]) {

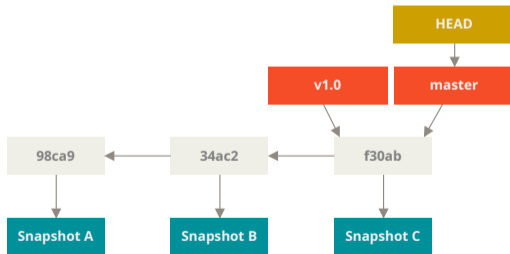
- if (argc = 1) {
+ if (argc == 1) {
     std::cout << "Not enough arguments\n";
 }
}
```

- If files are already staged, you can use `git diff --staged`

- Certain commits are of particular importance, e.g. bug fixes, new release, results for paper, etc.
- Git allows you to tag them to be able to refer to them easily.
- All the tag related actions are done using the `git tag` command. Use:
 - ▶ `git tag` to list all the tags,
 - ▶ `git tag <name>` to create a lightweight tag with name `<name>`,
 - ▶ `git tag -a <name> -m <message>` to create an annotated tag with name `<name>` and message `<message>`,
 - ▶ `git tag [-a] <name> <commit>` to create a tag on commit `<commit>`.

```
$ git log --oneline
a06ace4 (HEAD -> main, tag: v2.0.0, tag: Nature_2024_JDoe_et_al) Finalize release
11e5bed Add unit tests
87900b6 Improve documentation
0a2253b (tag: v1.1.0) Add feature A
e792bb1 (tag: v1.0.1) Fix bug
ce7cf28 (tag: v1.0.0) Initial commit
```

- One of the Git strengths is its ability to support nonlinear workflows.
- You can easily diverge from the main development line without impacting it.
- Git does that using *branches*.
- Before diving into branches, let's see how Git manages the references to commits.



- To isolate your work from the main development, create a branch using `git branch`:

```
$git branch testing
```

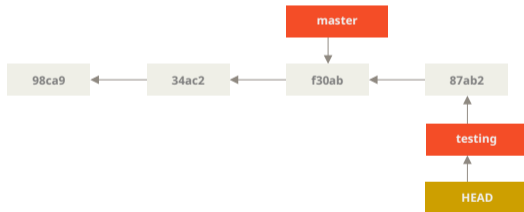


- Let's switch to the new branch using `git switch`:

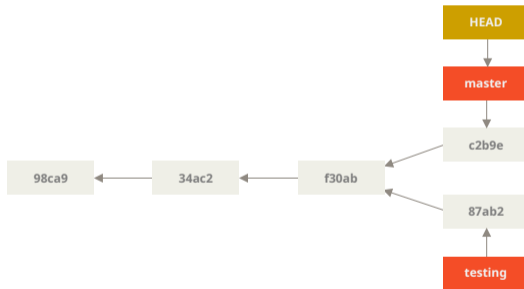
```
$git switch testing  
Switched to branch 'testing'
```



- We can now work on our branches separately without affecting the other.



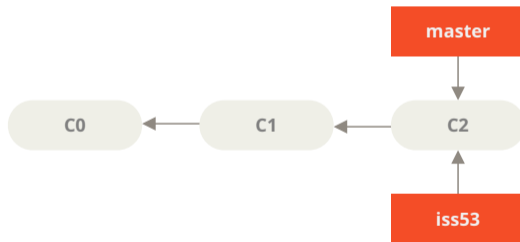
- We can now work on our branches separately without affecting the other.



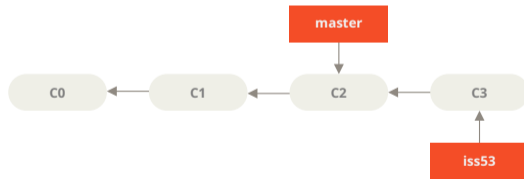
- Once you are satisfied with the work done on your branch, you can include it back to the main development branch.
- This is called merging. Use `git merge <branch to merge>`



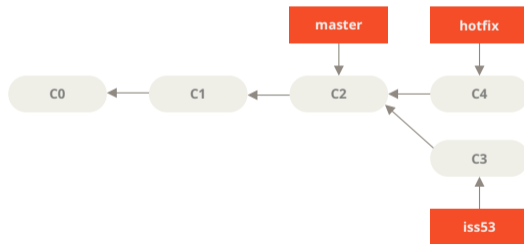
- Once you are satisfied with the work done on your branch, you can include it back to the main development branch.
- This is called merging. Use `git merge <branch to merge>`



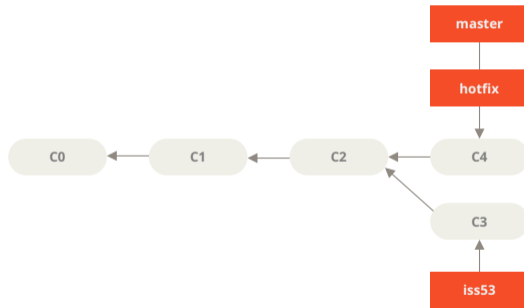
- Once you are satisfied with the work done on your branch, you can include it back to the main development branch.
- This is called merging. Use `git merge <branch to merge>`



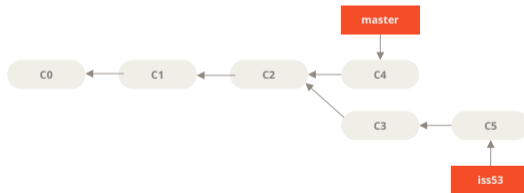
- Once you are satisfied with the work done on your branch, you can include it back to the main development branch.
- This is called merging. Use `git merge <branch to merge>`



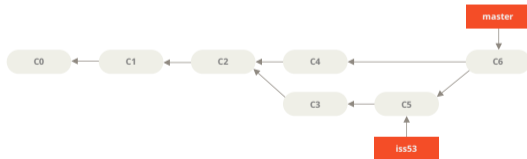
- Once you are satisfied with the work done on your branch, you can include it back to the main development branch.
- This is called merging. Use `git merge <branch to merge>`



- Once you are satisfied with the work done on your branch, you can include it back to the main development branch.
- This is called merging. Use `git merge <branch to merge>`



- Once you are satisfied with the work done on your branch, you can include it back to the main development branch.
- This is called merging. Use `git merge <branch to merge>`



- Reproduce the previous situation from scratch
 - ▶ Create 3 commits.
 - ▶ Add a new `iss53` branch, switch to it, and make a commit..
 - ▶ Switch back to `main`, create an `hotfix` branch and switch to it.
 - ▶ Merge `hotfix` into `main`.
 - ▶ Finally, merge `iss53` into `main`.

- On the `main` branch, create a file `data.txt` and write "This is the first line" in it.
- Commit this file.
- Create a new branch called `feature`.
- Still on `main`, add "This is a new line from main" into `data.txt` and commit it.
- Now, go to `feature` and add "This is a new line from feature" into `data.txt` and commit it.
- Merge `feature` into `main`.
- What happens? Use `git status` and inspect `data.txt` with an editor.

- You just witnessed a merging conflict!
- When there are changes in the same lines, Git doesn't know which one it should accept.
- You have to manually solve the conflicts!
 - ▶ Modify the regions marked with <<<<<<, =====, and >>>>>>.
 - ▶ Once everything is solved, type `git commit`.
- Options exist to tell Git what to do.

- Up until now, all the operations we have looked at are `local` to our repository.
- By its decentralized nature, Git allows you to share work with different special repos called `remotes`.

- The command `git remote [show <name>]` will print the information about the remotes.

```
$ git remote  
origin
```

```
$ git remote show origin  
* remote origin  
Fetch URL: git@gitlab.epfl.ch:SCITAS/courses/data-management-with-git.git  
Push URL: git@gitlab.epfl.ch:SCITAS/courses/data-management-with-git.git  
HEAD branch: main  
Remote branches:  
main          tracked  
Local branch configured for 'git pull':  
main merges with remote main  
Local ref configured for 'git push':  
main pushes to main (up to date)
```

- By default, the first remote is always called `origin`.

- You can add and remove remotes at your will with `git remote add/remove`.
- For example, you started a local remote and you want to push it to GitLab or Github.

```
$ git remote add <name> <repo address>
```

- Similarly, you may remove a repo:

```
$ git remote remove <name>
```

- To get updates from your remote, you can use `git fetch`.
- The command will download all the changes from a remote.
- It is important to note that `git fetch` will not merge the changes into your local copy.

```
$ git fetch origin
$ git log --graph --oneline --all
* 83a2e8e (origin/main) Add content
* a06ace4 (HEAD -> main, tag: v2.0.0, tag: Nature_2024_JDoe_et_al) Finalize release
* 11e5bed Add unit tests
* 87900b6 Improve documentation
* 0a2253b (tag: v1.1.0) Add feature A
* e792bb1 (tag: v1.0.1) Fix bug
* ce7cf28 (tag: v1.0.0) Initial commit
```

`git pull`

- To actually include the updates into your local copy, you can use `git pull <remote> <branch>`

```
$ git pull origin main
From <remote address>
 * branch          main          -> FETCH_HEAD
Updating a06ace4..83a2e8e
Fast-forward
file_5.txt | 1 +
1 file changed, 1 insertion(+)
```

```
$ git log --graph --oneline --all
* 83a2e8e (HEAD -> main, origin/main) Add content
* a06ace4 (tag: v2.0.0, tag: Nature_2024_JDoe_et_al) Finalize release
* 11e5bed Add unit tests
```

- In fact `git pull` is equivalent to `git fetch` followed by a `git merge`.

- At some point, you may want to share your work with your colleagues.
- To do so, you must push your changes to the remote using `git push <remote> <branch>`.

```
$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 260 bytes | 260.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
To <remote address>
a06ace4..83a2e8e  main -> main
```

- You can easily configure Git to make it easier to work with.
- There are three levels of configuration:
 - `--system` This will affect Git for all the users of the machine. The configuration is stored in `/etc/gitconfig`
 - `--global` This will only affect you and the configuration file is stored in `<home>/.git/config` or `<home>/.gitconfig`.
 - `--local` This will only affect the current repository. The configuration file is stored in `<project>/.git/config`.
- To set an option, use `git config --<level> <option> <value>`. For example:

```
$ git config --global user.name "John Doe"
$ git config --global user.email john.doe@epfl.ch
$ git config --global core.editor emacs
$ git config --global alias.graph log --oneline --graph --all
```

- Use `git config --list` to see your configuration.