THE DATA SCIENCE LAB

- Scalable Data Science with Python -

COM 490 – Module 1b

Week 2

EPFL

Agenda 2025 - Module 1a

1a	Introduction to Data Science with Python
1b	(Bigger) Data Science with Python
2a	Introduction to Big Data Technologies
2b	Big Data Wrangling with Hadoop
2c	Advanced Big Data Queries
3a	Introduction to Spark
3b	Spark Data Frames
3c	Advanced Spark
4a	Introduction to Stream Processing
4b	Stream Processing with Kafka
4c	Stream Processing with Kafka and Spark
Proj	Final Project, Q&A
Proj	Final Project Due (short video and code)
Proj	Oral Sessions

Quiz

- You are collecting data without a defined use case, which type of storage is most appropriate ?

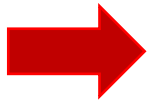
A. Data warehouse

B. Data lake

Data Lake or Data Warehouse – How to Decide

Aspect	Data Lake	Data Warehouse
Data Storage format	Raw: unstructured, semi-structured, structured (images, videos, audio, text, …)	Cleaned and structured
Schema Approach	**Schema-on-read** - structure applied at query time (flexible, schema evolution)	**Schema-on-write** – predetermined structure enforced before storage, limited flexibility
Storage Cost	Cost-effective, scalable storage	Higher storage cost (optimized, proprietary format)
Performance	Good for flexible, exploratory analytics, machine learning, scientific computing	High performance for standardized business intelligence queries
Use Case	Big data, Machine Learning, data discovery & exploration, ad-hoc analytics	Business Intelligence, interactive dashboards and reporting
Governance	Requires external tools to maintain quality (⚠ effort required to avoid data swamp)	Strong built-in quality control and data catalog
Typical Users	Data scientists and engineers	Business analysts and decision makers

Data Lake versus Data Warehouse – Key Takeaway



- **Data Lake** : ideal when you want flexible storage of all kinds of data and will define structure as you explore it.

- **Data Warehouse** : ideal when you have well-defined, structured data use cases and need fast, reliable query performance for BI.

Why is Data Lake Storage Flexibility Important ?

Because you control how data is stored, you can:

- Optimize storage and query costs by choosing the right **Data Format**
- Easily adapt to changing needs without being locked in
- Store many types for data flexibly

Data Storage Format – What Is It ?

A data format defines how data is organized and stored on disk or exchanged (transmitted over a network), including:

- How records, rows, or columns are laid out
- How values are encoded - text, binary, …
- How the structure of the data is represented – headers, indexing …

Data Storage Format – Why It Matters ?

Different data formats have trade-offs across several dimensions

- **Storage efficiency** : disk usage and cost
- **Read/write performance** : latency and resource requirements (CPU, I/O)
- **Structure flexibility** : how easily the format adapts to changes in the data
- **Data exchange** : interoperability between systems and protocols

Key takeaway - choosing the right format depends on:

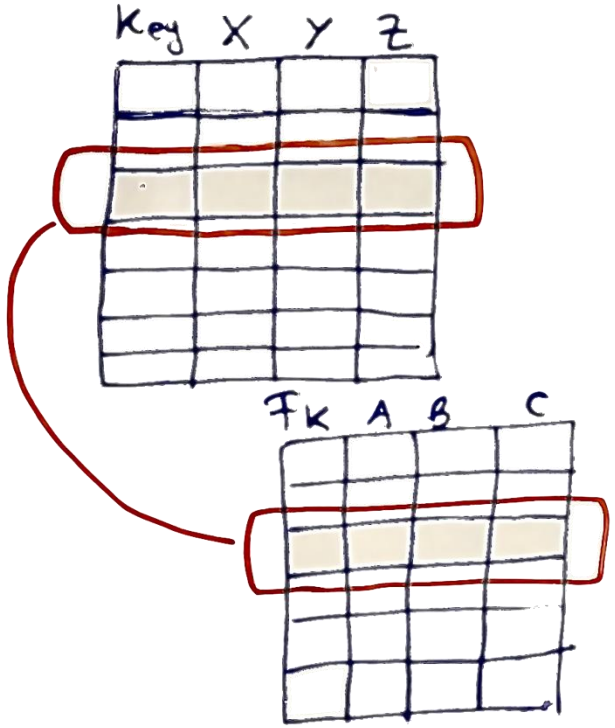
- Use case: **OLTP** vs **OLAP** vs Machine Learning & Scientific Analysis, vs data exchange
- **Technology stack**: Hadoop, Spark, relational database, data pipelines, …

OLTP Versus OLAP – What Are They ?

Transactional Systems:

OLTP (Online Transaction Processing)

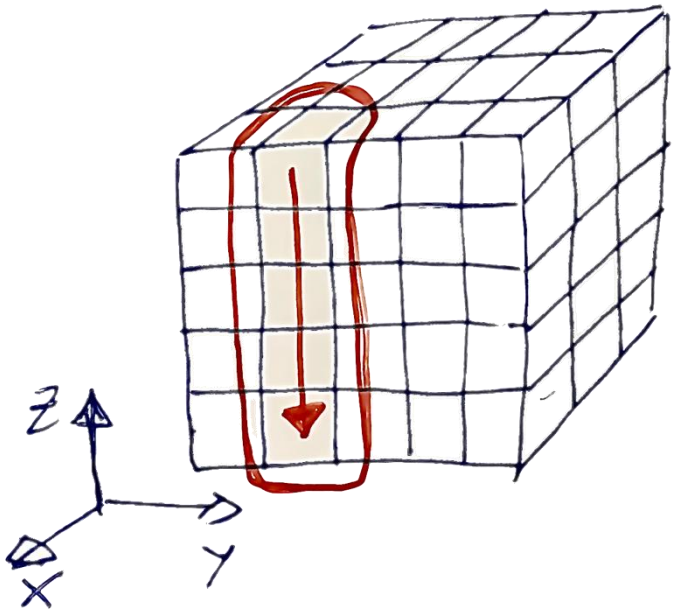
Best on Row-based storage



Usually normalized for write efficiency and data consistency; efficient for retrieving and updating specific records. E.g. HR systems to manage company employees.

Analytical Systems:

OLAP (Online Analytical Processing)



Best on Columnar-storage

Usually denormalized (flat) for read performance (avoid expensive table join) and optimized for columnar analytics and multidimensional aggregations.

OLTP versus OLAP versus Scientific Analysis

Aspect	OLTP	OLAP	ML / Scientific Analyis
Purpose	Transactional Systems	Analytical & reporting	Multidimensional science, numerical arrays, meta data rich (self describing)
Typical Query	Short, fast single/few row access; insert, update, delete.	Large aggregations across dimensions; bulk scans for business analytics	Machine Learning, scientific computation; slicing multidimensional arrays
Data Storage	Row-based: rows stored sequentially	Columnar-based: columns stored sequentially.	Multidimensional arrays, optimized for slicing (read)
Example formats	CSV, JSON, XML (legacy), Database tables	Parquet, ORC, …	NetCDF4, HDF5, images, …

Data Format – How To Decide

Data Type	Purpose	Recommended format	Why?
Big Tabular 2 Data	OLAP, Business Analytics	PARQUET, ORC	Compact, efficient analytics.
Small scale, structured or semi structured	Data exchange, enriching other data sets e.g. sensor network configuration.	CSV (tabular), JSON (nested)	Simple, portable, human readable, at small scale sometime more efficient than other formats.
Scientific, multidimensional >= 3D	Machine Learning, Scientific computing, and sharing of scientific data.	HDF5, NetCDF	Optimized for large arrays, self-describing with units, scales.

Note: those are the more popular formats due to their portability. They are other types of course, such as AVRO which is best for data exchange/serialization with schema evolution, …

Python Data Processing Frameworks …

Machine Learning Libraries
Scikit-Learn
PyTorch, TensorFlow (DL)
Optuna (hyperparameter tuning), …

DataFrame Libraries
Pandas, Modin
Vaex, Polars
DuckDB, …

Specialized DataFrame Libraries
GeoPandas (for geospacial analytics)
Xarray (pandas for high dimensional data), …

Distributed Engines
Dask
Ray
PySpark, …

Data Exchange Libraries
PyArrow
Petastrom, …

To name a few …

Advanced Analytics Libraries: Built for Big Data, Beyond Pandas

Why Choose Advanced Analytics Libraries over Traditional Ones like Pandas?

- **Out-of-core** processing: Handle datasets larger than your memory by processing data in manageable chunks instead of loading it all at once.
- **Partitioning & parallelism**: Split data for faster processing across multiple CPU, enabling efficient query execution.

Bottom line

- When your dataset no longer fits comfortably in memory, these libraries should be your first choice.

In-Memory Versus Out-of-Core Processing

In-Memory

- Entire dataset (or at least the part being processed) fits into memory (RAM) during computation
- Faster, once data is in memory
- Limited by available RAM
- Used when data is small
- E.g. **Pandas**

Out-of-Core

- Data is loaded in **chunks** into memory during processing
- More **Disk/Network I/O**: needed to retrieve chunks of data
- RAM just need to be big enough to contain a chunk.
- Used when data set exceeds RAM
- E.g. **Vaex**, **Polars**, **DuckDB**,

Understanding Partitioning In Data Storage

- **Organizes data** into partitions based on column values, e.g., year, month, day

```
…/weather/year=2025/month=01/day=01/*.parquet
                              /day=02/*.parquet
                              …
                    /month=02/day=01/*.parquet
```

- Can be used in **parallel processing** to further optimize queries

E.g. `SELECT COUNT(*) FROM table GROUP BY year`

Groups are counted in parallel

- Can be used in **predicate pushdown** to further optimize queries

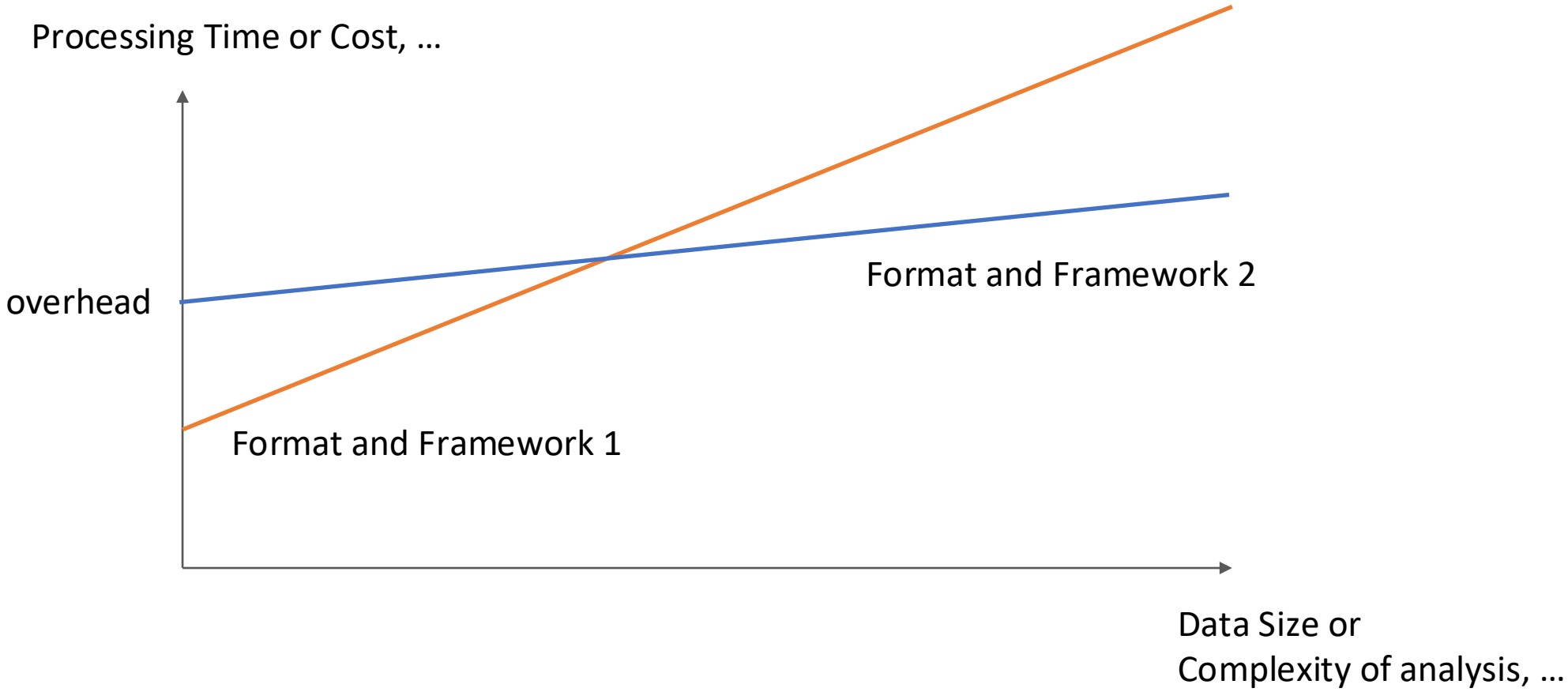
  Reduces I/O by skipping irrelevant partitions during queries

E.g. `SELECT … FROM table WHERE year=2025 AND month>6`

(1) Example shows a hive partitioning

Data Formats and Python Frameworks – Key Takeaway

There will be tradeoffs !



In Conclusion - How Shall I Start the Data Journey ?

- ~~Easy! I'll go ahead and start building ML models, right?~~ Wrong!
- Instead, start by…
  - Ingesting data
  - Storing, cleaning & integrating data (data format, frameworks)
- In most companies, this actually represents **75%** of the work
- Only then can you make the last **25%** (*analytics*) successful
- Build a **data platform** to tame your data first !

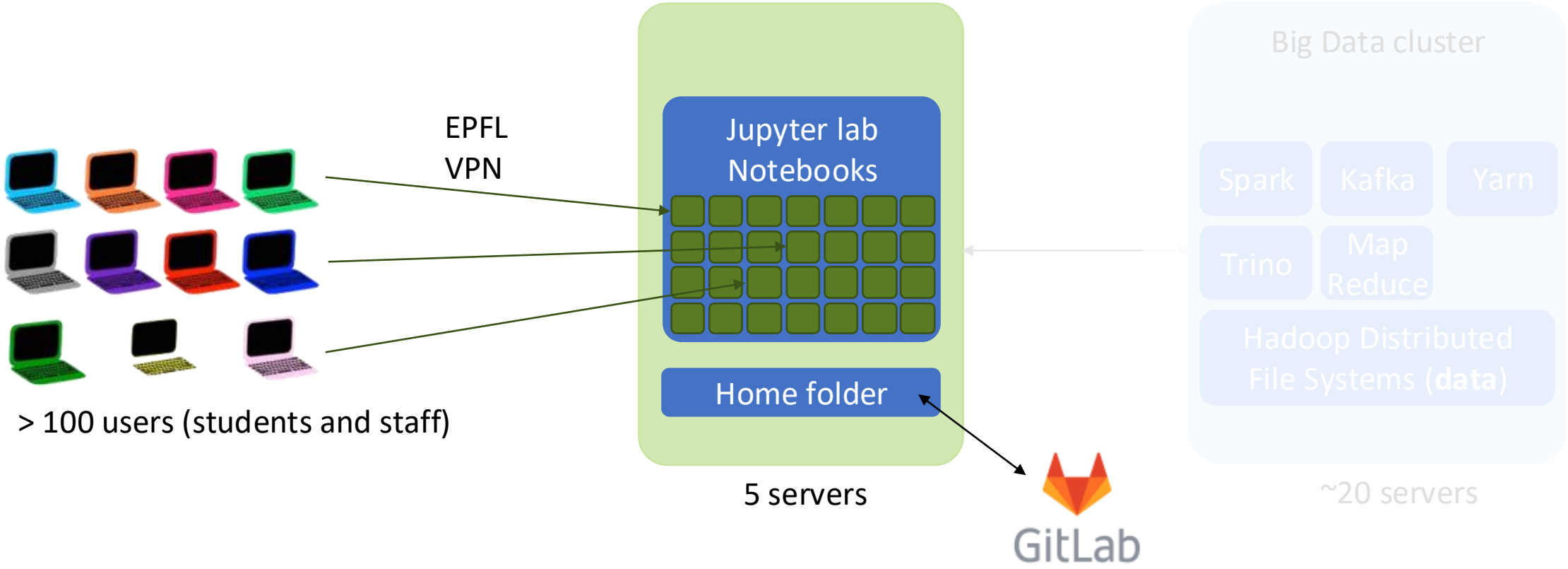
Today's check list – key objectives

- **Most of you have formed the groups**
  - Otherwise contact us
- **You have access to the gitlab**
  - You can login to https://dslabgit.datascience.ch/
- **You understand the purpose of git and master the *most commons* commands**
- **You should be able to determine an efficient data storage format for your needs**
  - Or at least avoid an obviously less efficient storage for the purpose
- **You are aware of different python data processing technologies available to you**
  - And understand the tradeoffs.
- **You become familiar with those terms**
  - Data warehouse vs data lake, out-of-core vs in-memory, OLAP vs OLTP vs scientific processing, partitioning, parallel processing

Start Your Engines
- Lab -

https://dslabgit.datascience.ch/course/2026/module-1b

Programming Environment



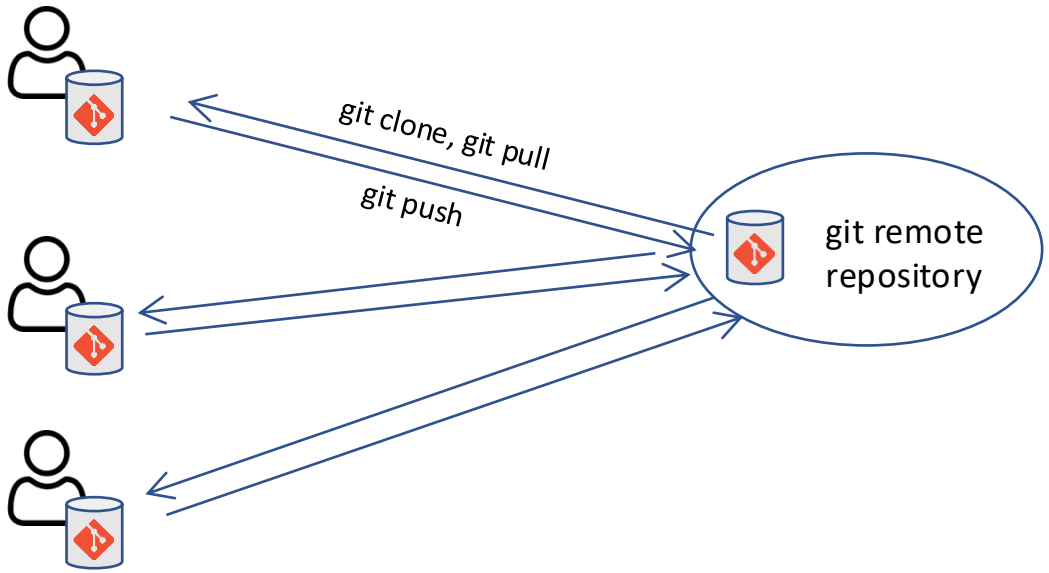
1. **BYOL**: Students work remotely using their laptops. Nothing to install – only web browser is needed.
2. Students work in teams, write and share code and environment in jupyter notebooks and gitlab
3. All data stored, and compute intensive processing executed on the distributed Big Data cluster.

Appendix
Code Versioning with Git

Crash Course

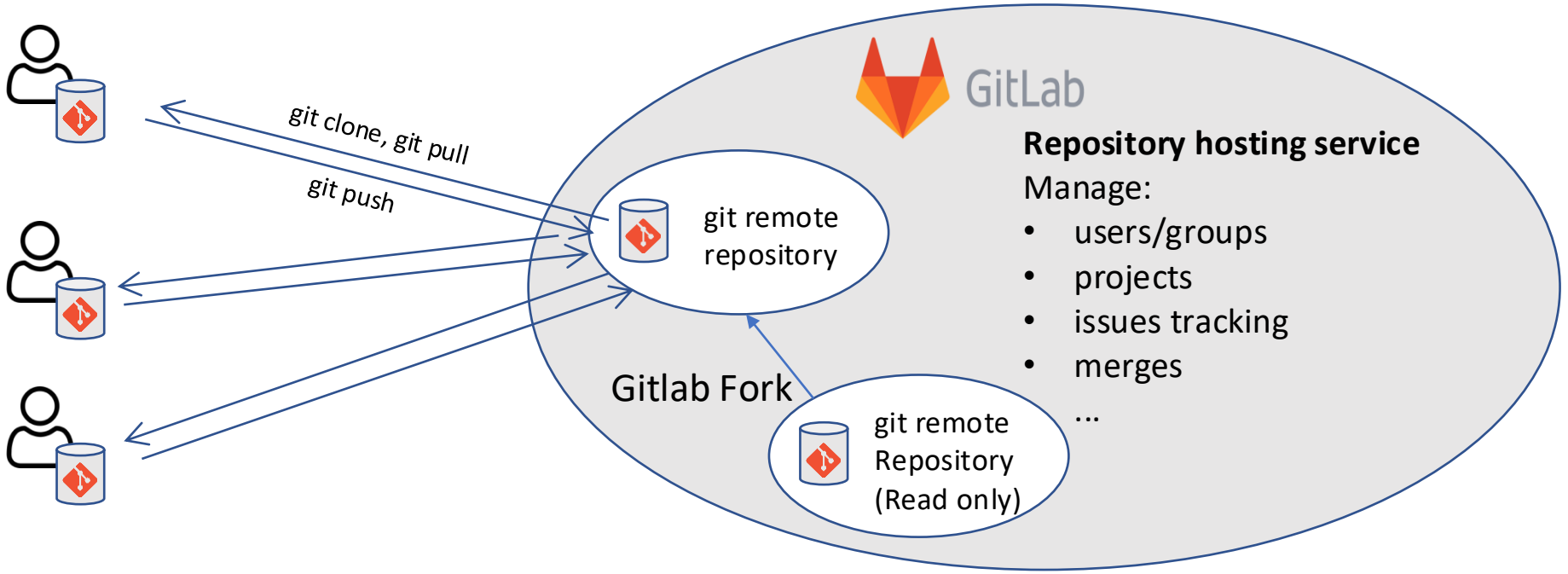
Git – distributed code version system

- *Tracks changes in computer files, used for coordinating collaborative work among programmers*
  - Created in 2005 by Linus Torvald, now used for 95% of version control tasks

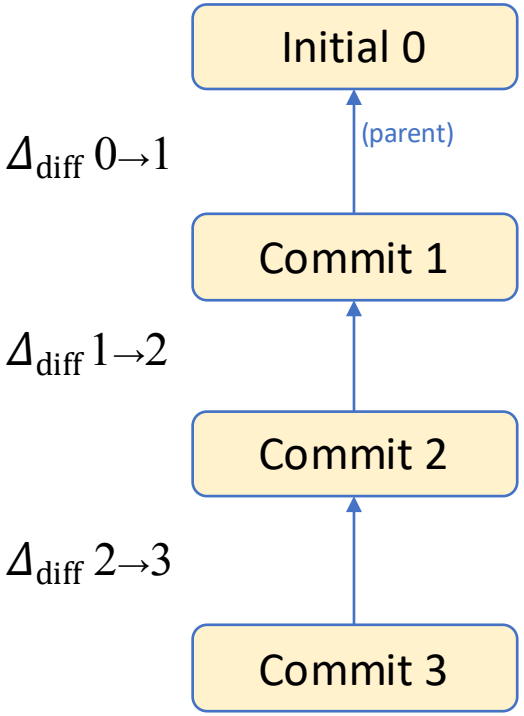


Git – distributed code version system

- *Tracks changes in computer files, used for coordinating collaborative work among programmers*
  - Created in 2005 by Linus Torvald, now used for 95% of version control tasks



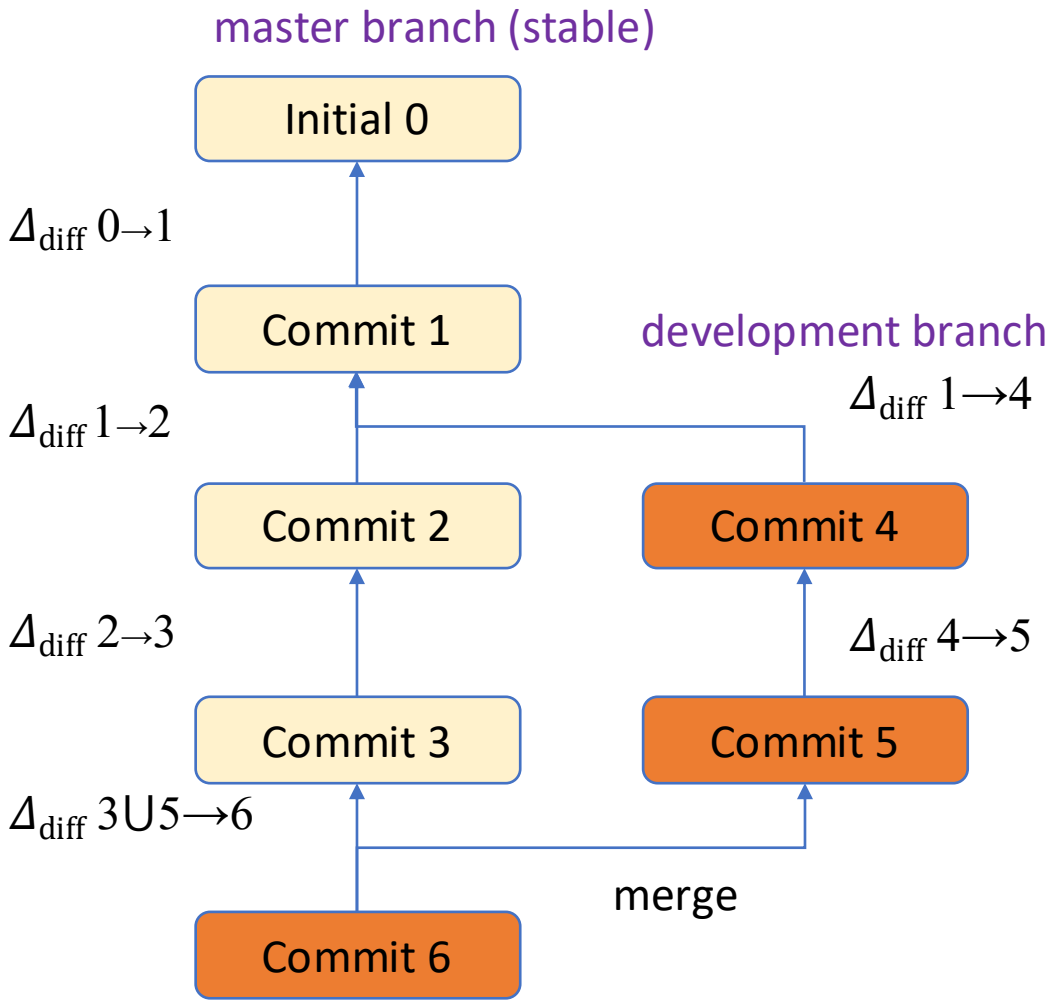
Git – Commit History (git log)



- A **commit** is a single point in the commit history. It is a snapshot of all the tracked files at that point.
- A commit is identified by a unique ID, e.g. d1f5510… which is derived from the content of that snapshot
- By default, commits form a linear history

source: git-scm.com

Git – Branching



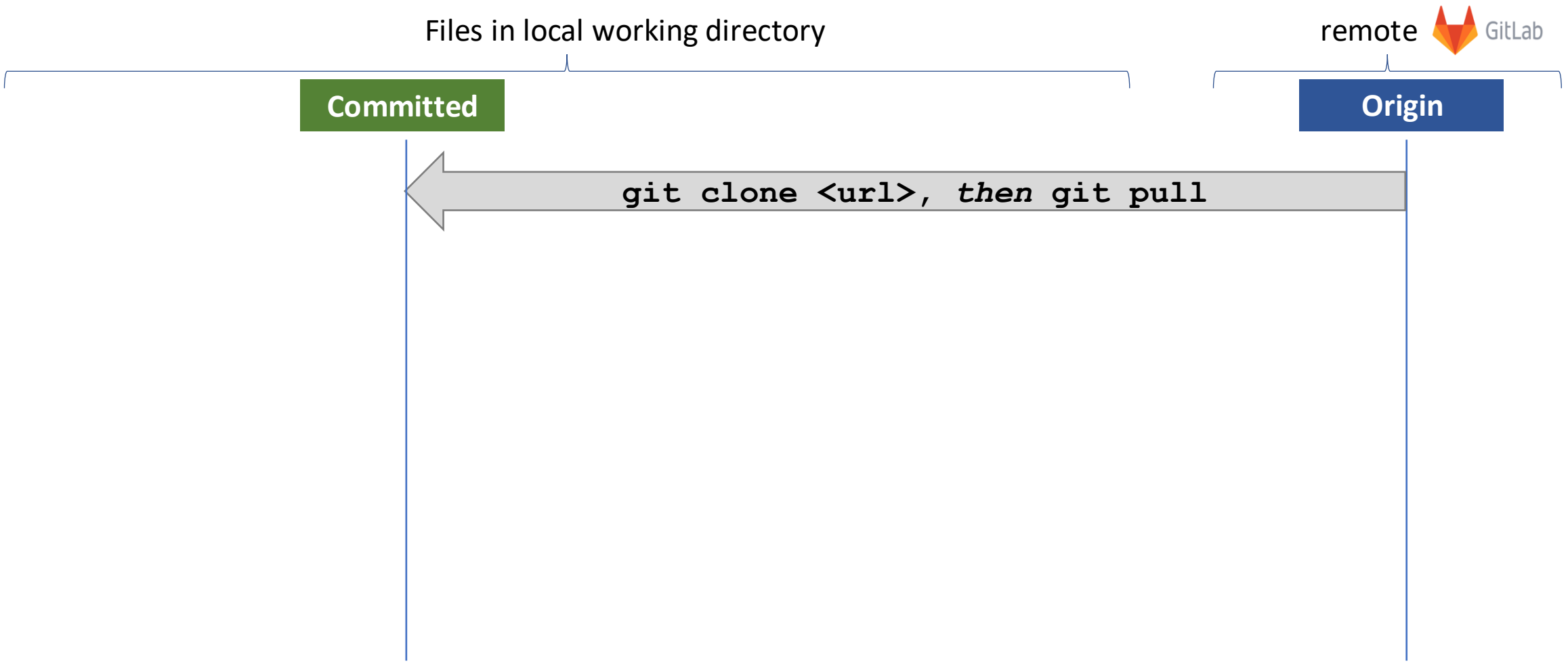
- A **commit** is a single point in the commit history. It is a snapshot of all the tracked files at that point.
- A commit is identified by a unique ID, e.g. d1f5510… which is derived from the content of that snapshot
- By default, commits form a linear history
- Using Git, it is recommended to work in parallel on separate branches (e.g. stable **main** branch and development branches)

Git branching strategies

- GitFlow (complex)
- Github Flow (easy)
- Gitlab Flow (easy)
- OneFlow (medium)
- …

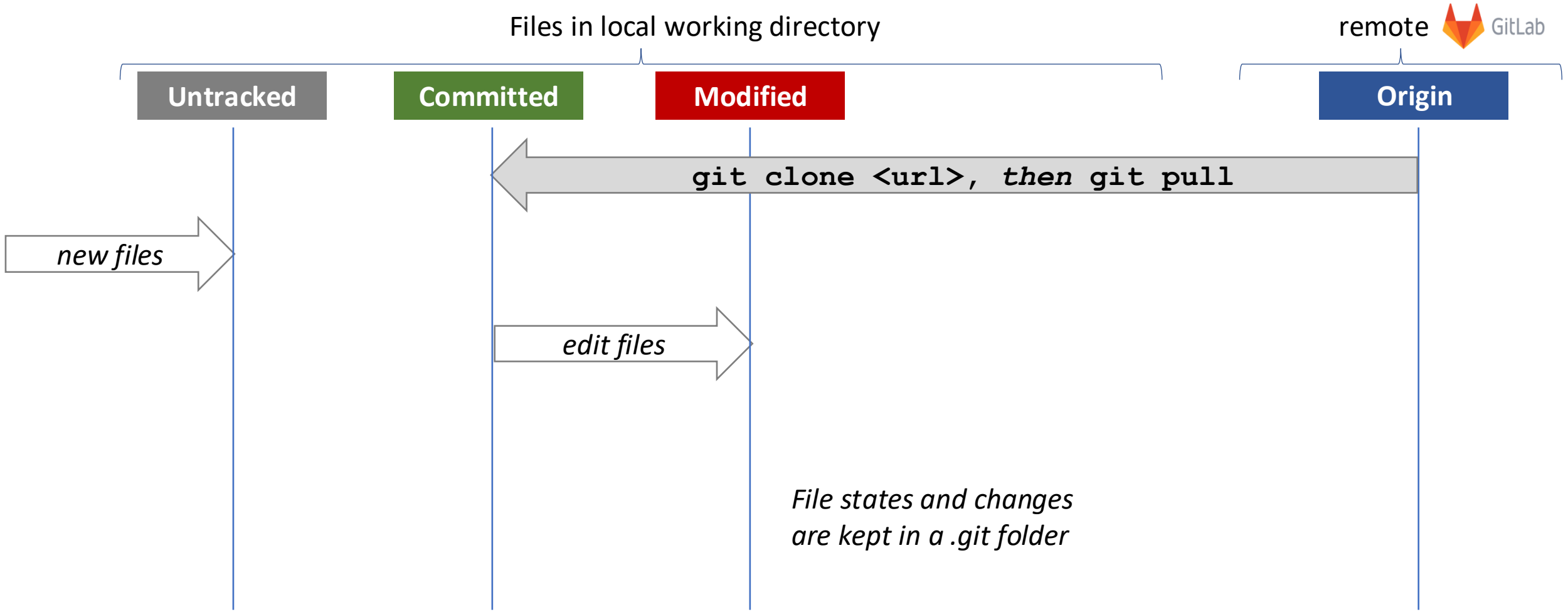
source: git-scm.com

Git – File Lifecycle



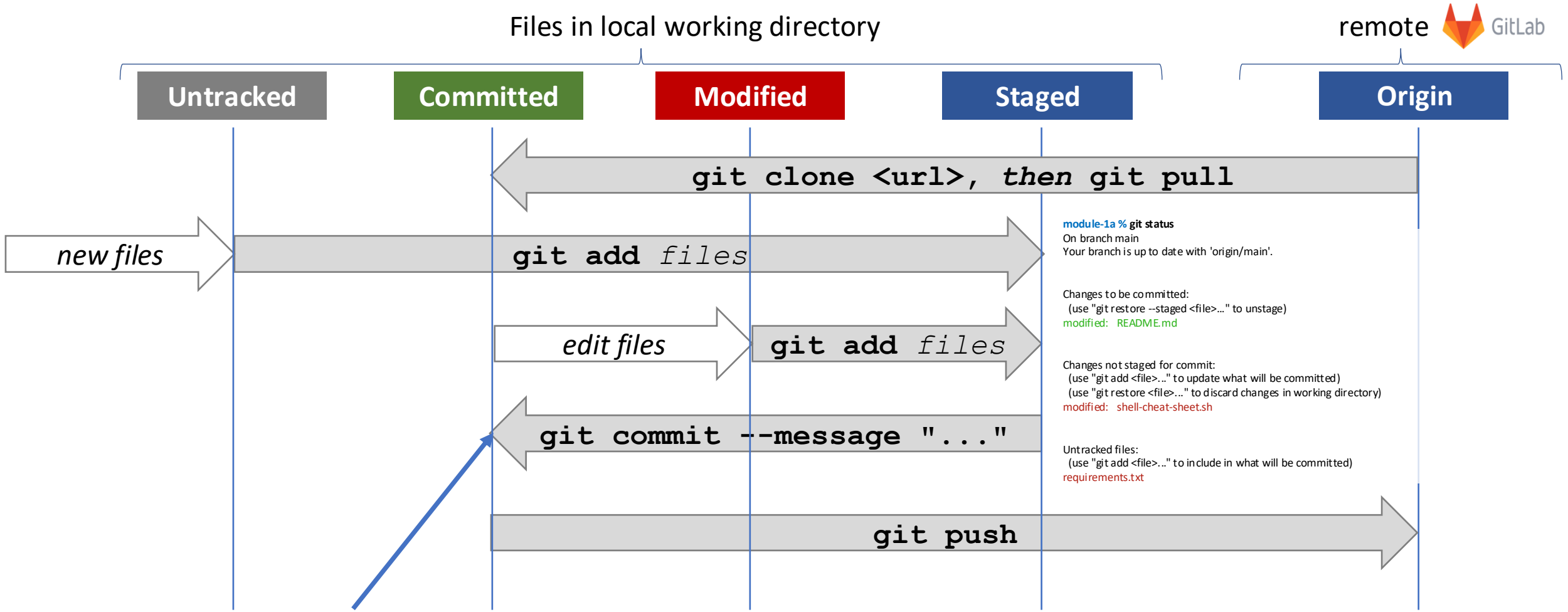
source: git-scm.com

Git – File Lifecycle



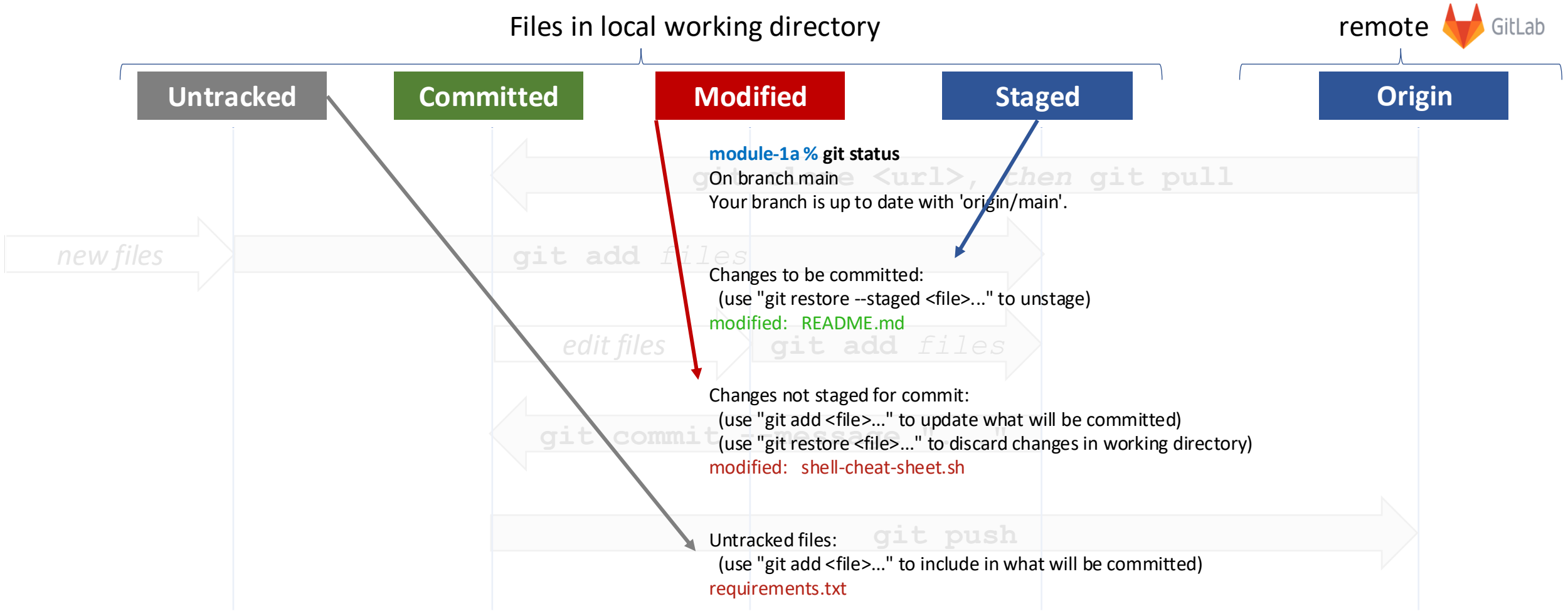
source: [git-scm.com](git-scm.com)

Git – File Lifecycle



source: [git-scm.com](git-scm.com)

Git – File Lifecycle



Git – Common Commands

- Copy a repository locally from a remote origin

```
git clone https://com490-2024.epfl.ch/com490-2024/module1b.git
```

- Stage files for a grouped commit

```
git add files
```

- Commit files from staged area

```
git commit --message "description courte de la validation"
```

- Verify status of files repository (untracked, modified, staged files, …)

```
git status
```

- Push local changes to remote (origin) repository

```
git push
```

- Retrieve changes from remote (origin) repository

```
git pull
```

Git – Common Commands

staged commits

- Display commit log (project history)

```
git log --all --graph
```

```
git log --stat -M
```

- Checkout earlier commit (or start a new branch)

```
git checkout [-b new-branch-name] {commit-id | branch-name}
```

- Show manual

```
git help command
```

- Move file

```
git mv file-from file-to
```

- Stop tracking a file

```
git rm --cache file
```

```
git reset file
```

```
git restore --staged file
```

Git – Less Common Commands

staged commits

- Unstage file and keep changes, or undo last n commits

```
git reset file
```

```
git restore --staged file
```

```
git reset HEAD^n
```

- Undo changes to a file

```
git checkout -- file
```

- Show current changes (difference)

```
git diff [HEAD~n|commit-id|branch-name] [file]
```

- Integrate changes between branches, rewrite history

```
git rebase -i [commit-id|branch-name]
```

Git – Best Practices (for starters)

- **DO NOT** commit large files
  - If needed use git lfs (see appendix)
- **DO NOT** commit binaries, intermediate data files, notebooks with outputs, or any files that can be regenerated from source
- **DO** use separate branches for 'clean' code and 'development' code
  - `git checkout -b yourname-dev`
  - `git add` & `git commit`
  - `git push --set-upstream origin yourname-dev`
  - Merge branch *yourname*-dev to main branch in gitlab (after review)
- **DO** use short but meaningful commit messages (wip, bufix, …)
  - https://www.conventionalcommits.org/en/v1.0.0/

Gitlab Setup

Before using gitlab you need to setup your credentials (ssh keys) so that you can authenticate with our gitlab service.

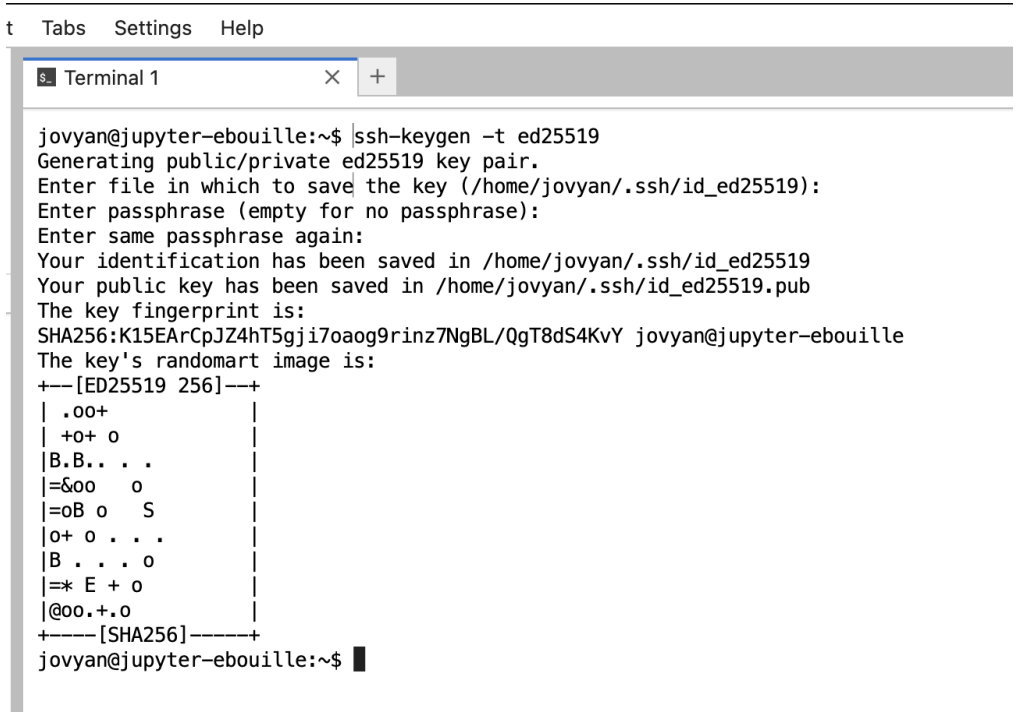
- This should be done automatically the first time you log in Jupyter Hub
- You should have received an email about a new key (com490) being added to your gitlab profile
- There should be a private ssh key in your home folder: `ls ~/.ssh/com490_{username}_{year}_key`
- An ssh-agent (key chain) should be running under your name: `pgrep -u $USER ssh-agent`

If you are still being asked for a password when using git, read-on this **Gitlab Setup**

Gitlab Setup

Before using gitlab you need to setup your credentials (ssh keys) so that you can authenticate with the gitlab service

1. Sign in to your assigned jupyter hub server iccluster***.iccluster.epfl.ch
2. In a terminal enter the command: `ssh-keygen -t ed25519`
3. Press enter to each prompt



Gitlab Setup

The last command should have created a private and a public ssh keys in the folder ~/.ssh

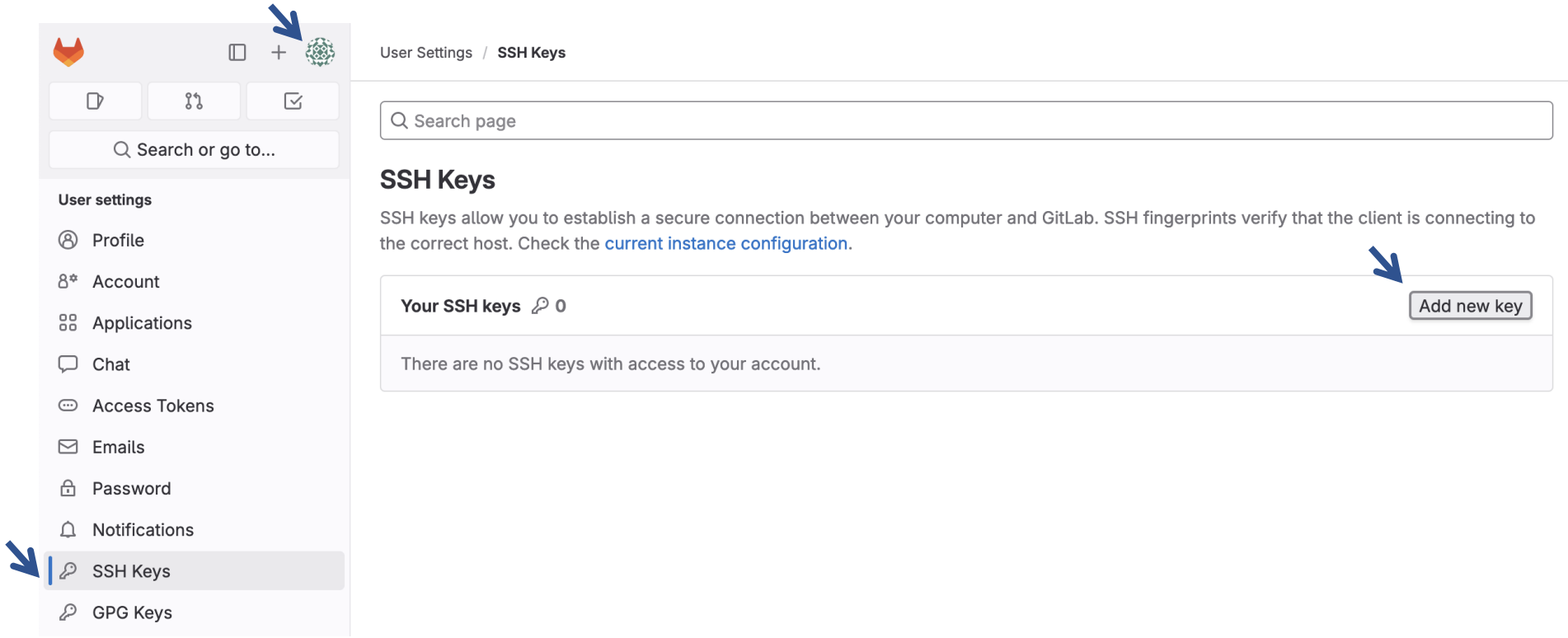
1. In a terminal enter the command to display the public key (.pub): `cat ~/.ssh/id_ed25519.pub`
2. Select and copy the content of the public key

Gitlab Setup

1. Sign in to gitlab

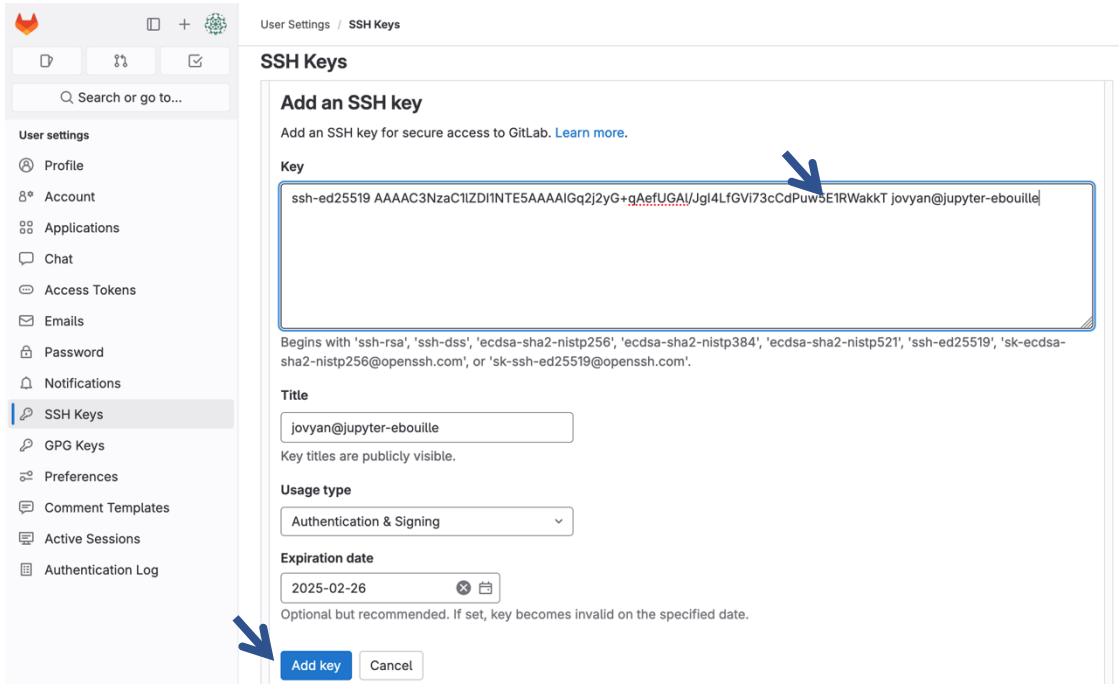
   You should have received an invitation email to set up your password, if expired you can request a new invitation.

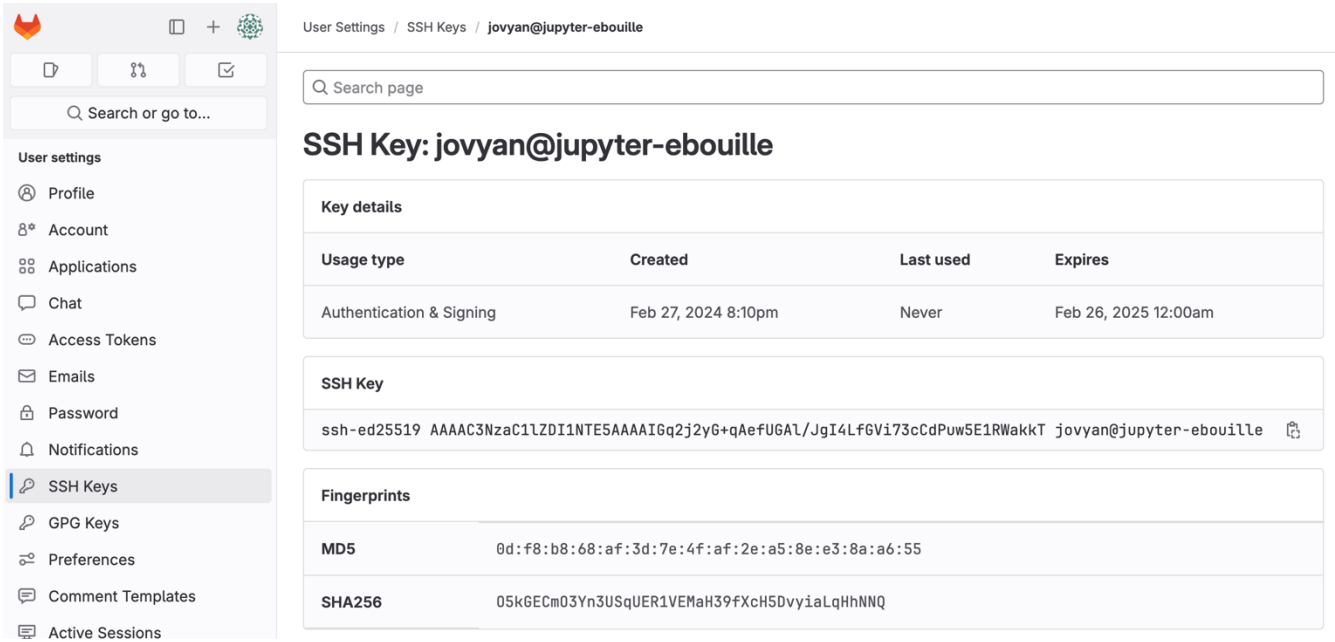
2. In your profile (click on the avatar), select '**SSH keys**', and '**Add new key**'



Gitlab Setup

Copy the public key and click '**Add key**' to save

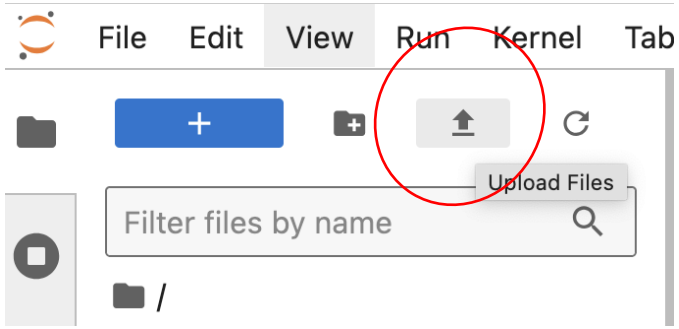




Gitlab Setup

In the same folder ~/.ssh

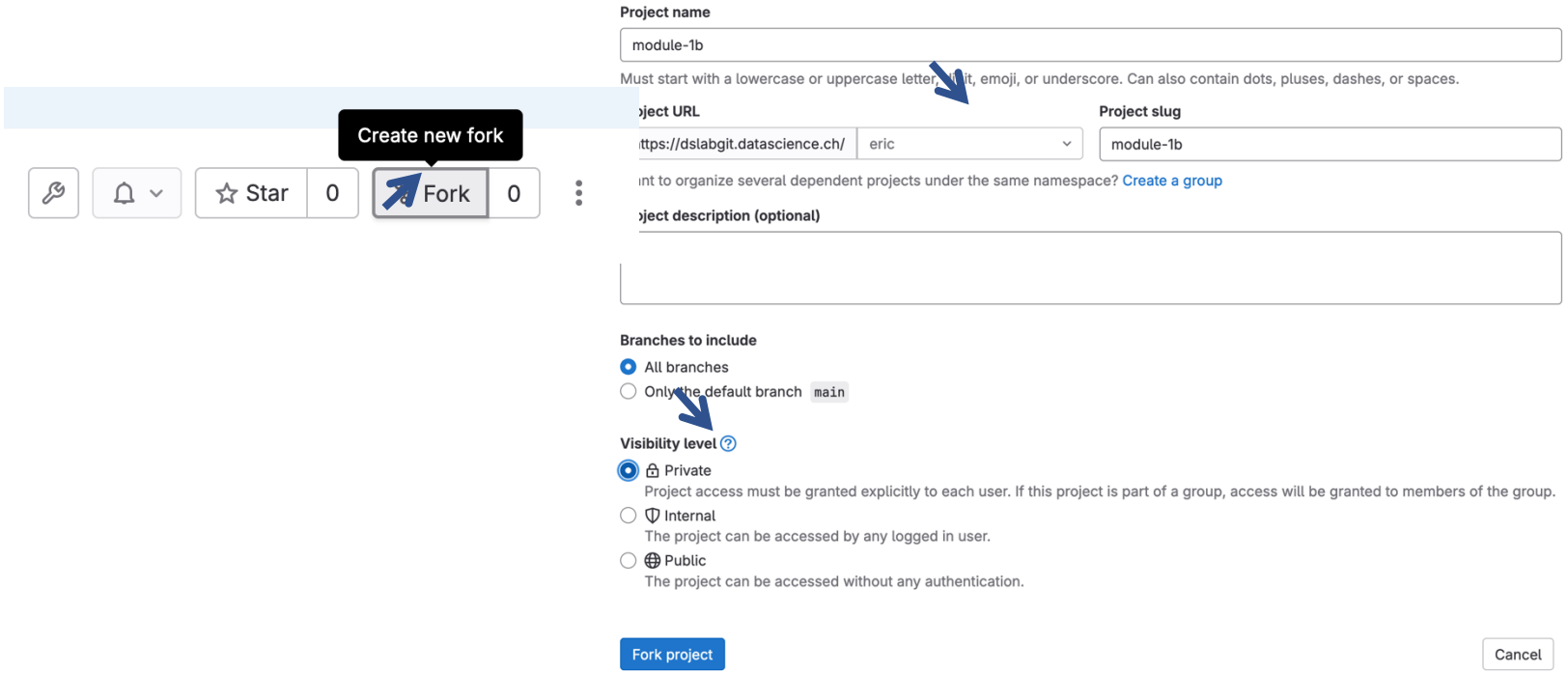
- Create or edit the file ~/.ssh/config
  - Or if this is easier create the file locally and upload it in Jupyter Hub.
- Add the following lines to it (id_ed25519 is the private key) and save




```
Host dslabgit.datascience.ch
    HostName dslabgit.datascience.ch
    User git
    IdentityFile ~/.ssh/id_ed25519
    IdentitiesOnly yes
```

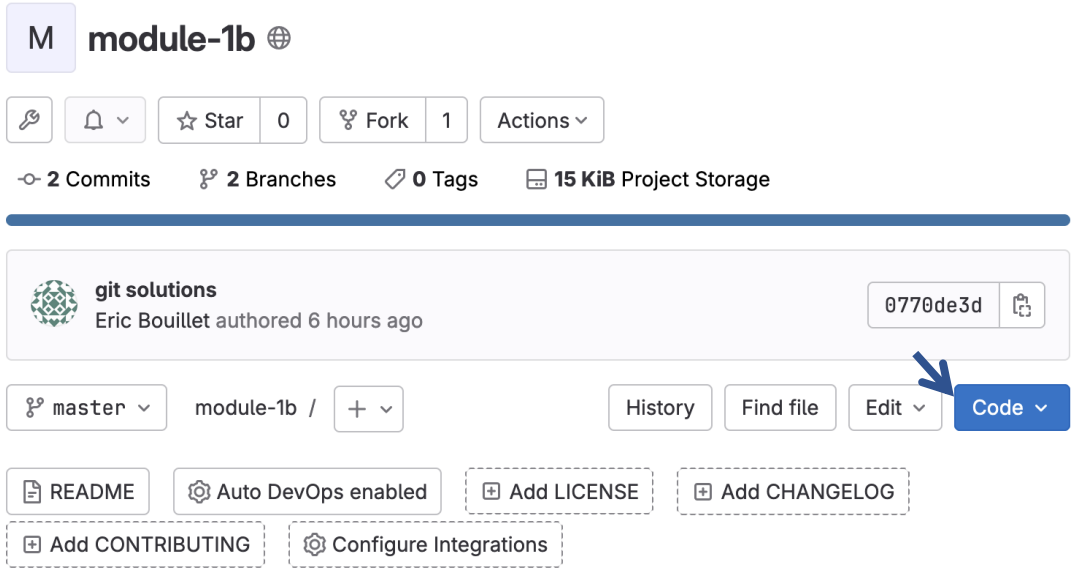
Gitlab – Making a copy (fork) of a git repository

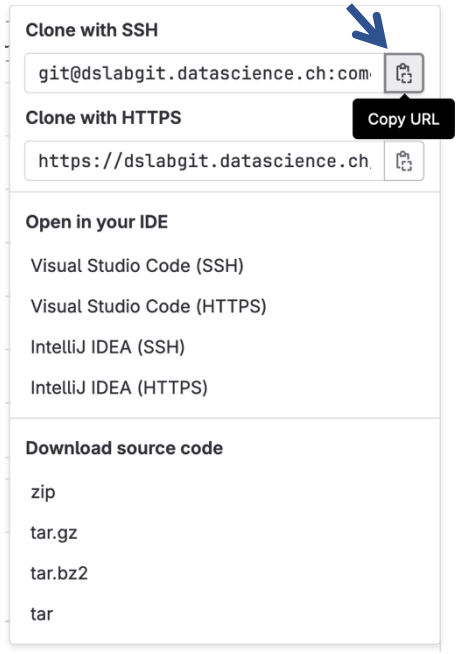
1. Sign in to gitlab and navigate to the project you want to copy
   E.g. https://dslabgit.datascience.ch/course/2025/module-1b
2. Fork the project, under your name or gitlab group name (e.g. /students/2025/A1), set the visibility to private



Clone your git repository

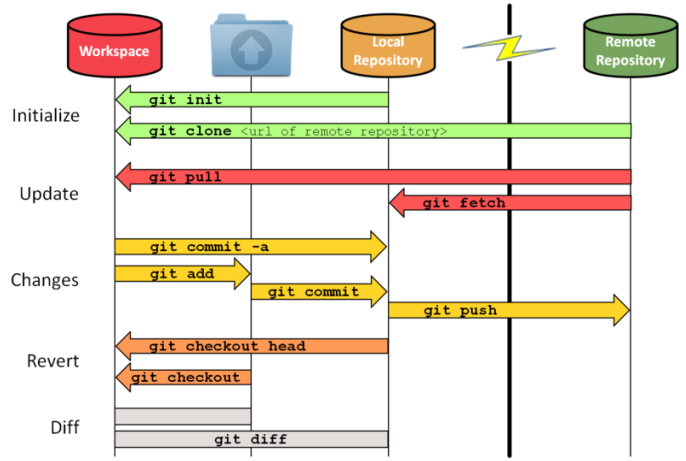
- In **gitlab** open your copy (after fork) of the git repository, and in 'Code' copy the git@dslabgit.datascience.ch:<repository>.git URL





- In a **Jupyter** terminal, enter the command: `git clone` *paste-URL-you-just-copied*

Git – Documentation



1. https://git-scm.com/docs
2. https://education.github.com/git-cheat-sheet-education.pdf
3. https://docs.gitlab.com/ee/topics/git/
4. Terminal tutorial

More Useful info in Appendix