THE DATA SCIENCE LAB
Bigger Data Science with Python

COM 490 – Module 1b

Week 2

Agenda 2025 - Module 1b

19.02	Introduction to Data Science with Python
26.02	(Bigger) Data Science with Python
05.03	Introduction to Big Data Technologies
12.03	Big Data Wrangling with Hadoop
19.03	Advanced Big Data Queries
26.03	Introduction to Spark
02.04	Spark Data Frames
09.04	Advanced Spark
16.04	Introduction to Stream Processing
30.04	Stream Processing with Kafka
07.05	Advanced Stream Processing
14.06	Final Project Q&A
22.05	Final Project Videos Due before midnight
28.05	Oral Sessions

Agenda

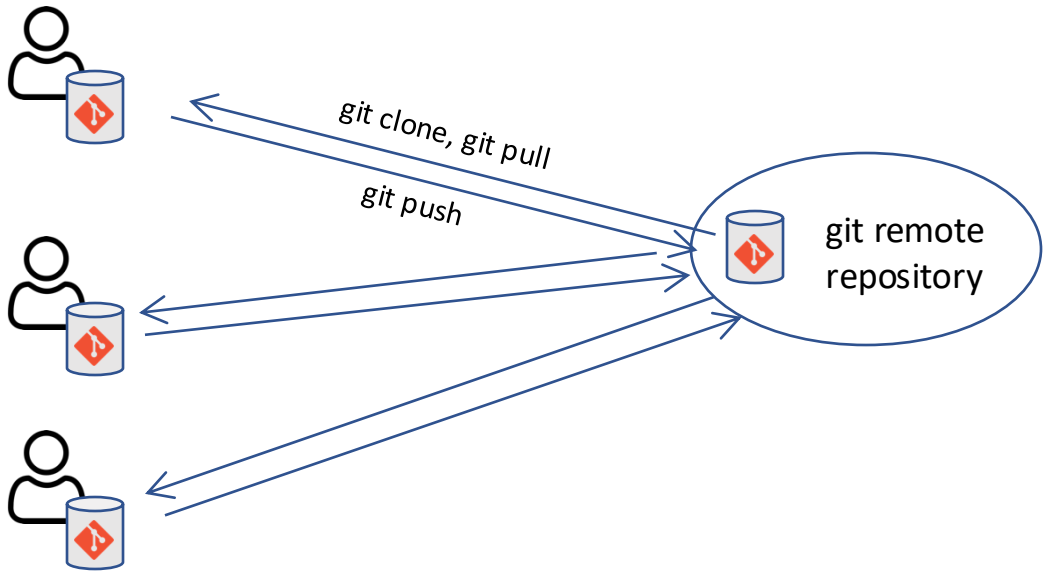
- Code versioning with Git
- Data Storage Formats
- Python – Scalable & Parallelized Data Processing Frameworks

Code Versioning with Git

Crash Course

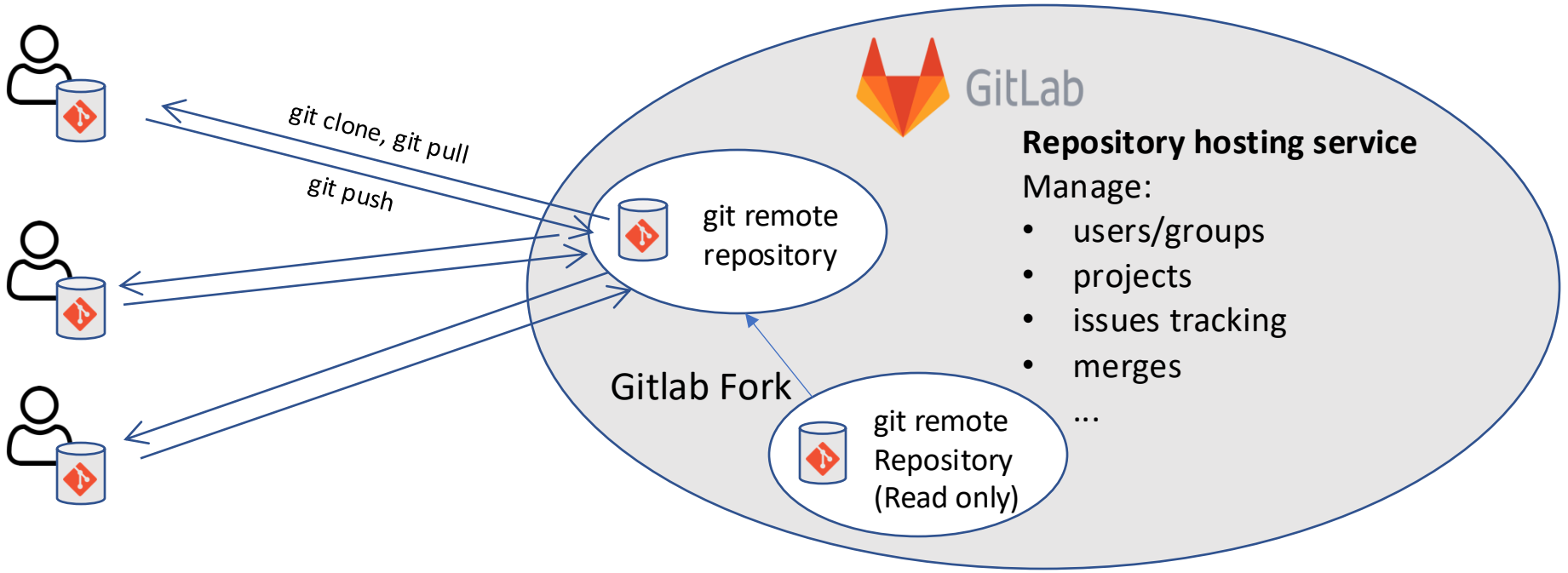
Git – distributed code version system

- *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

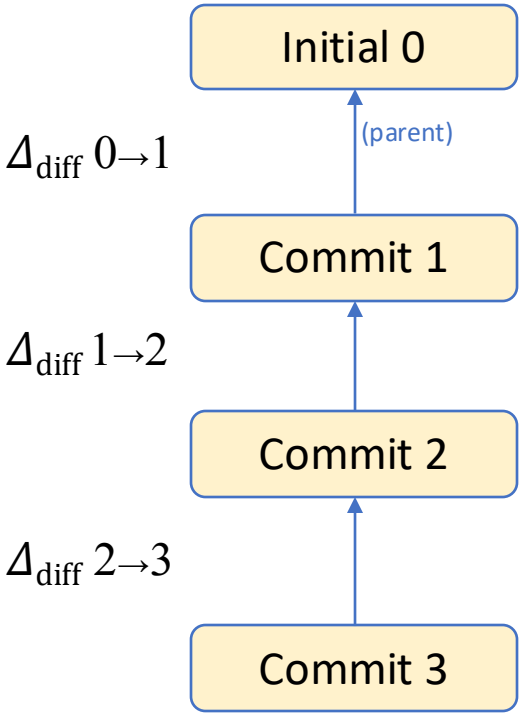


Git – distributed code version system

- *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



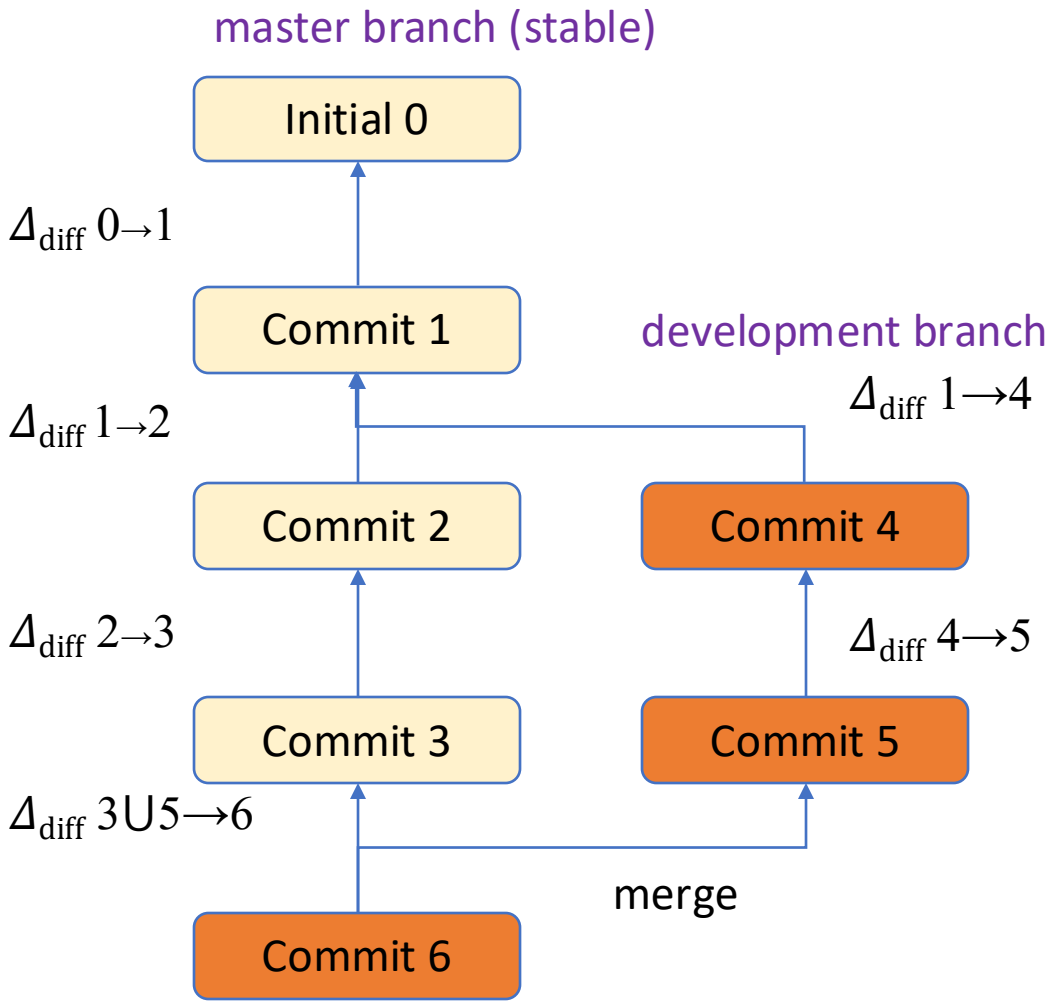
Git – Commit History (git log)



- A commit is a single point in the commit history. It is a snapshot of all the tracked files at that point.
- By default, a succession of commits follow a linear evolution

source: git-scm.com

Git – Branching



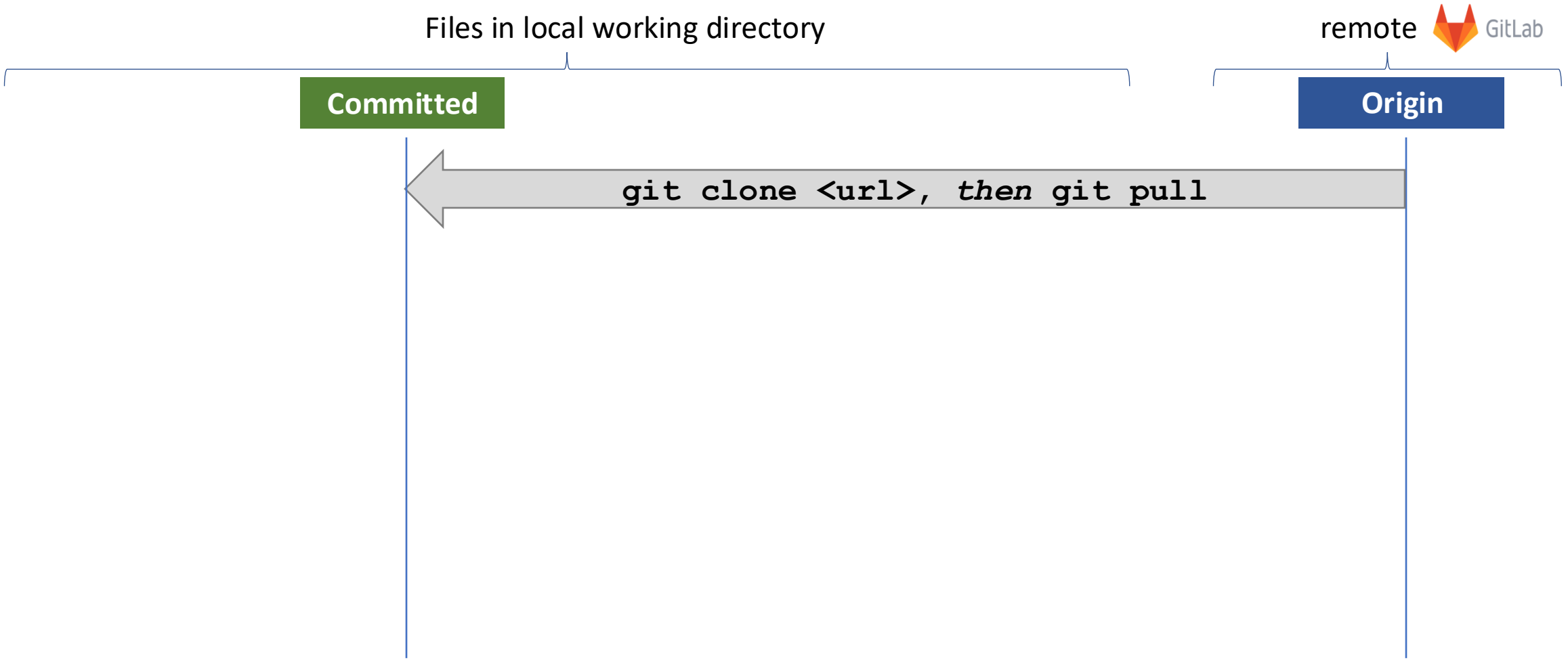
- A **commit** is a single point in the commit history. It is a snapshot of all the tracked files at that point.
- By default, a succession of commits follow a linear evolution
- Using Git, it is recommended to work in parallel on separate branches (e.g. stable **master** branch and development branches)

Git branching strategies

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

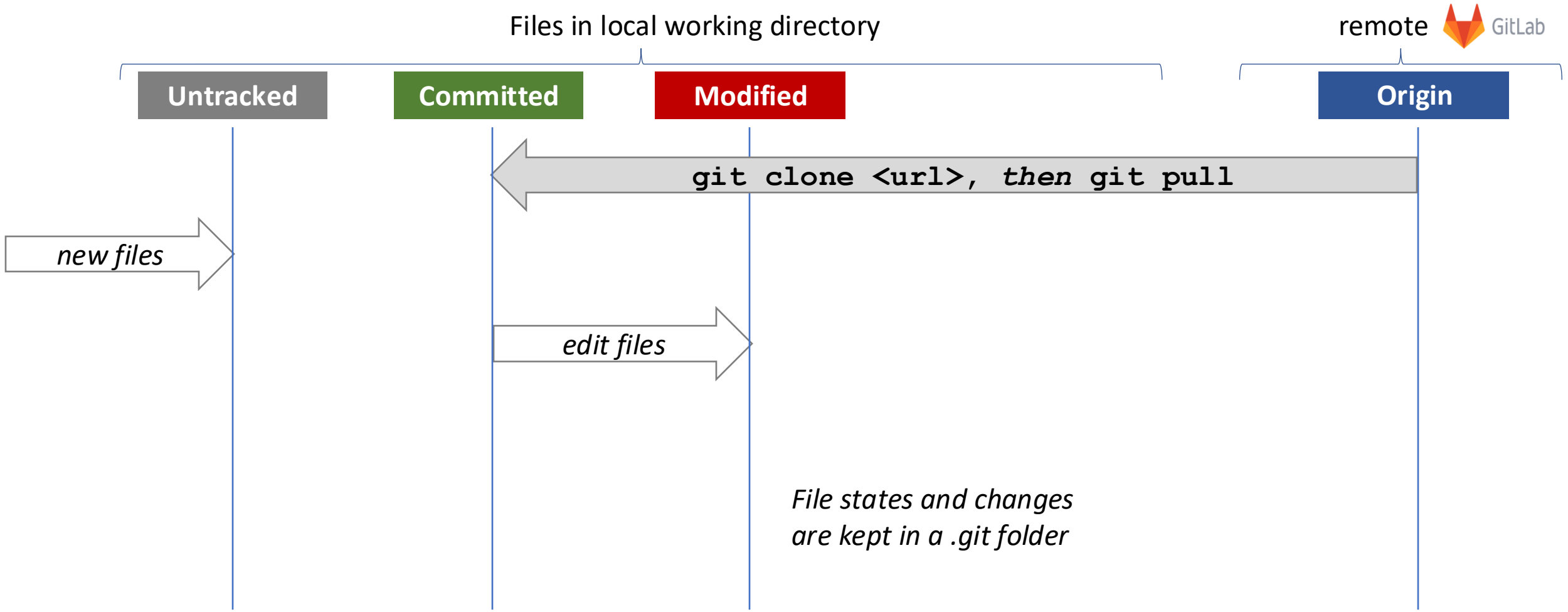
source: git-scm.com

Git – File Lifecycle



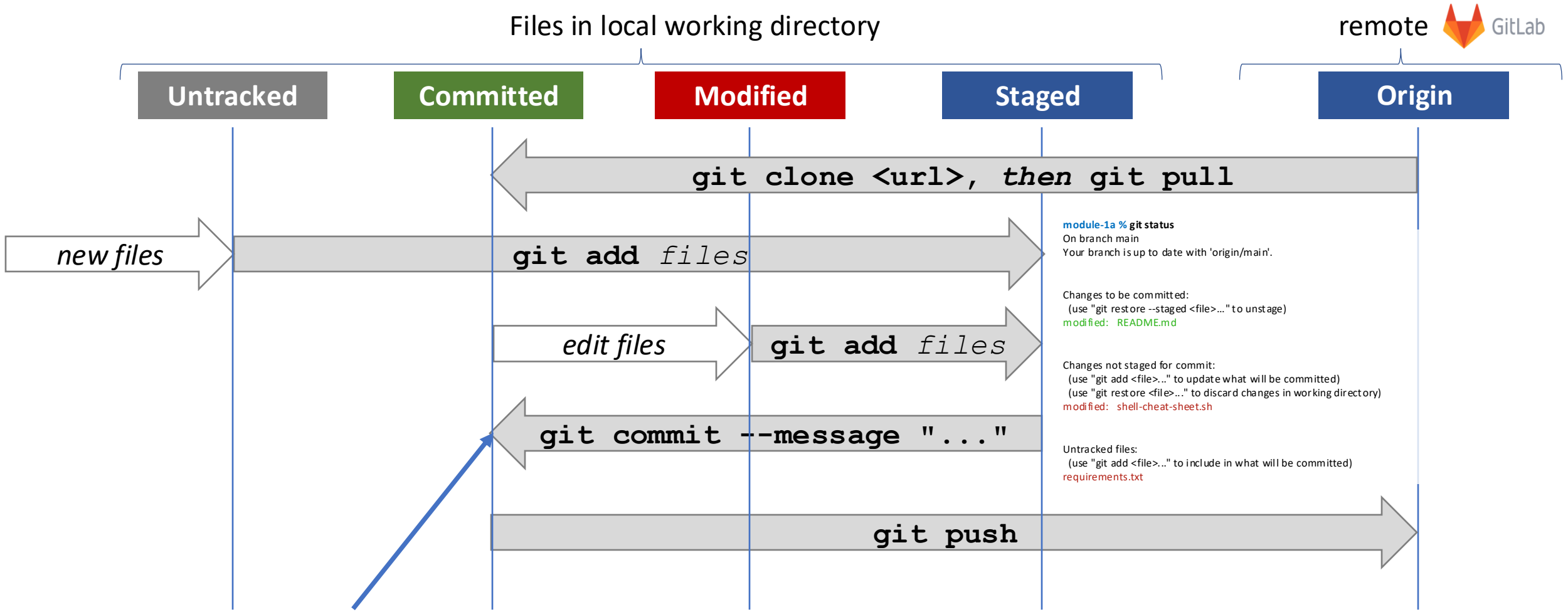
source: [git-scm.com](git-scm.com)

Git – File Lifecycle



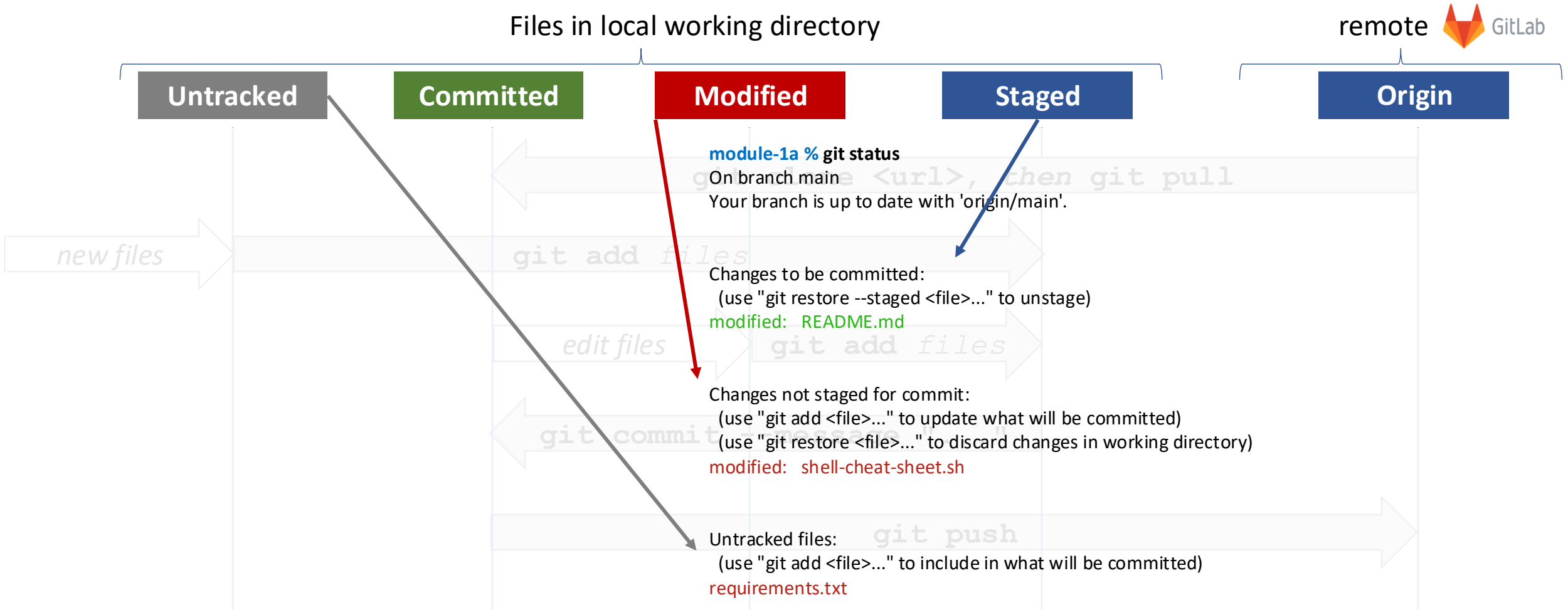
source: [git-scm.com](git-scm.com)

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
```

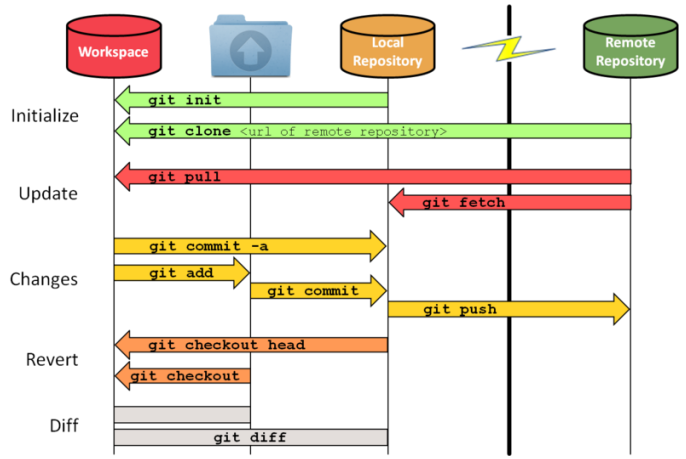
- Retrieve changes from remote (origin) repository

```
git pull
```

Git – 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/

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

Data Storage Formats

Data Storage Formats - Introduction

- Different data formats offer trade-offs
  - Storage efficiency (cost of storage)
  - Schema flexibility (usability)
  - Read/write performance (latency, and cost of query RAM/CPU req.)
- Popular formats(*)
  - CSV, JSON, (XML), …
  - Avro
  - Parquet, ORC, Feather
  - HDF5, NetCDF
- Choosing the right format depends on use cases like **OLTP**, **OLAP**, machine learning, or scientific data analysis and technology used

(*) There are many other formats optimized for very specific applications (e.g. ONNX for DL, DICOM for imaging etc), we only discuss multi-purpose format here

Understanding Data Processing – OLTP vs OLAP

OLTP (Online Transaction Processing)

- **Purpose:** Handles real-time transactional data (e.g., banking transactions).
- **Data Type Format:** Relational databases with structured data (e.g., SQL databases like MySQL or PostgreSQL).
- **Key Characteristics:** Fast, high-volume, **row-based**.

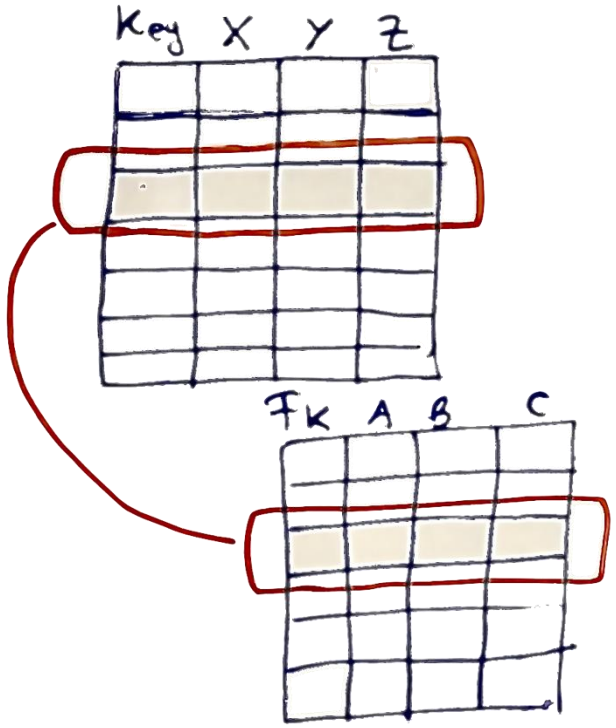
OLAP (Online Analytical Processing)

- **Purpose:** Analyzes large datasets (e.g. performance metrics).
- **Data Type Format:** Data warehouses and multidimensional databases
- **Key Characteristics:** Large-scale, **columnar storage**.

Understanding Data Processing – OLTP vs OLAP

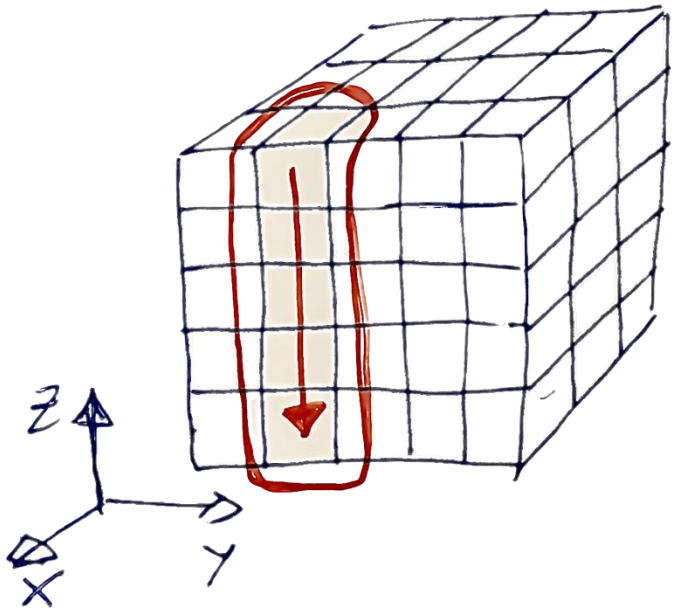
OLTP (Online Transaction Processing)

Row-based



Normalized for consistency

OLAP (Online Analytical Processing)

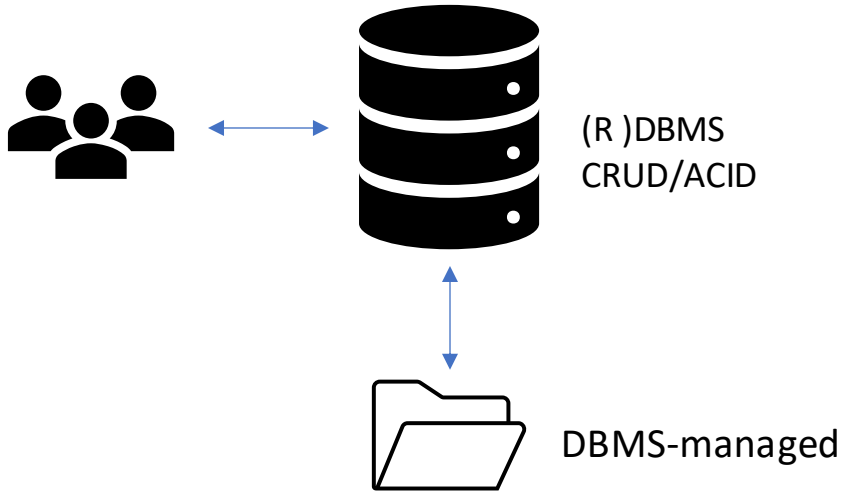


Columnar-storage

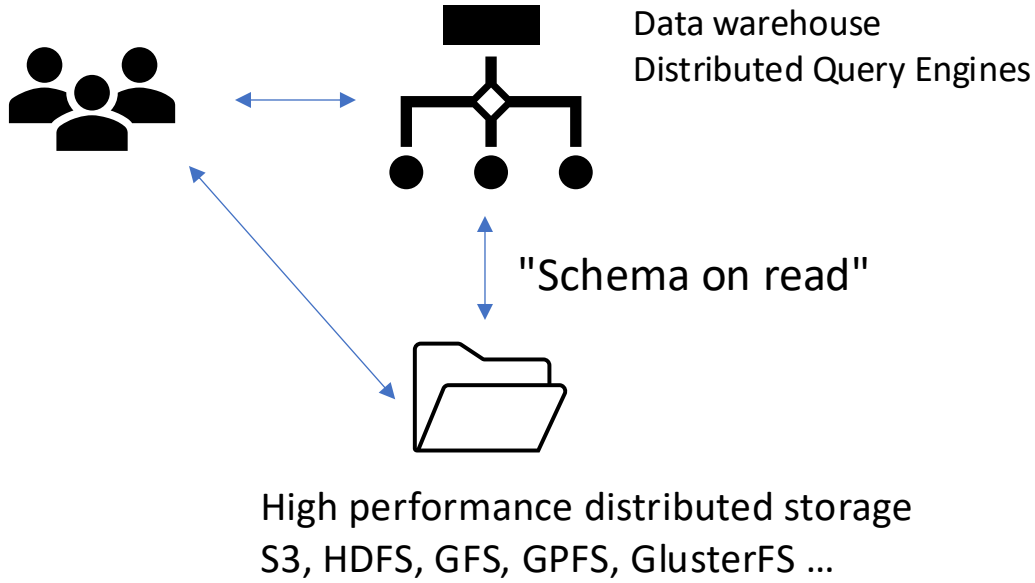
Operations on columns or slices along dimensions
Denormalized for faster querying and complex analysis

Understanding Data Processing – OLTP vs OLAP

OLTP (Online Transaction Processing)



OLAP (Online Analytical Processing)



(Typical approaches to OLTP and OLAP)

Data Storage Formats – Key Considerations

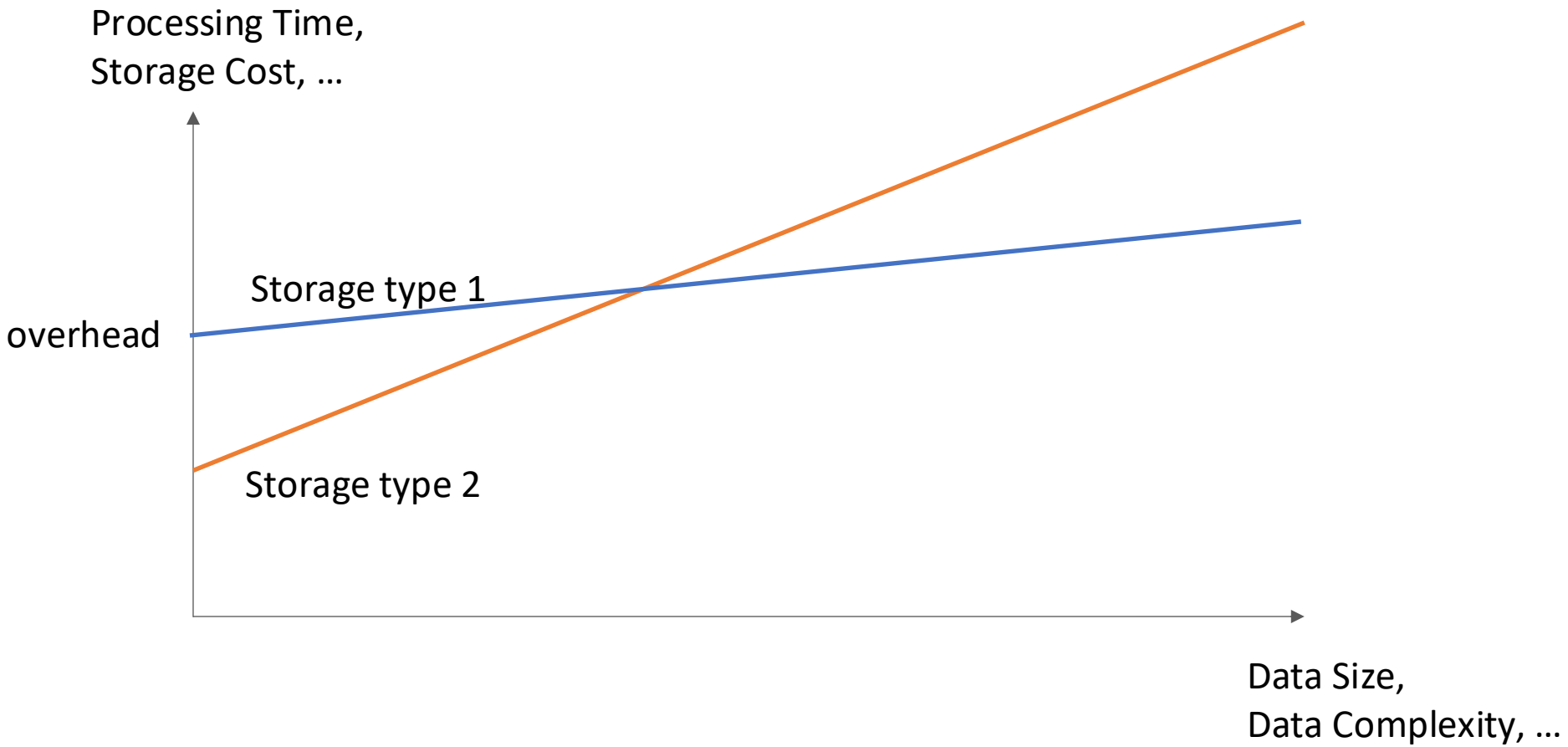
DATA TYPE	RECOMMENDED FORMAT	BEST FOR	WHY?
Tabular (Big Data)	**PARQUET** or **ORC** (or **Feather**)	OLAP	Compact, fast reads, columnar storage, indexing
Semi-structured / Nested	**Avro**	OLTP, streaming	Compact, fast serialization, *schema evolution*
Scientific, multi-dimensional	**HDF5** or **NetCDF**	OLAP	Optimized for large arrays, fast random access, metadata-rich
Small-scale or Human Readable	**CSV** (tabular), **JSON** (nested)	N/A (manual small data)	Simple, portable, human-readable, slow reads (text parsing)

See Appendix for a detailed comparison

Data Storage Formats – Key Takeaways

- For the purpose of analytical queries, if you receive data in
  - CSV
    - Convert to Parquet or ORC for faster analytical queries and smaller storage footprint
  - JSON or XML
    - Tabular
      - Convert to Parquet or ORC
    - Nested or hierarchical:
      - HDF5/NetCDF for scientific/multidimensional data

Data Storage Formats – Tradeoffs



Python Data Processing Libraries

Python Data Processing Libraries

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

DataFrame Libraries
Pandas, Modin
Vaex, Polars
DuckDB, …

Specialized DataFrame Libraries
GeoPandas
Xarray, …

Distributed Engines
Dask
Ray
PySpark

Data Exchange Libraries
PyArrow
Petastrom, …

Data Frame Libraries - Key Considerations

Tool	Scalability	Typically Used With	Key Features
Pandas	Single machine, slow	CSV, Parquet	Convenient, polyvalent …
Polars	Single machine, extremely fast, lazy data loading (reduced RAM overhead)	Parquet, CSV, JSON	Multi-threaded, Rust-based, lazy evaluation
Vaex	Single machine, out-of-core (lazy data streaming)	Parquet, CSV, HDF5	Memory-mapped, no RAM overhead
Modin	Single machine (e.g. unitdist MPI) or distributed on Dask, Ray, …	Parquet, CSV, JSON	Pandas "drop-in" replacement for parallelized Code
DuckDB	Single machine, SQL-based, out-of-core, very fast (great for 100Gb)	Parquet, CSV, JSON	In-memory OLAP, local analytics

"Scalable pandas replacements"

Specialized Data Frame Libraries

Tool	Scalability	Integrate Well With	Typically Used With
GeoPandas	Geospatial Vector Data (2D)	Dask, Pandas	Shapefiles, GeoJSON, Parquet
xarray	Multidimensional (N > 2)	Dask, Pandas	HDF5, NetCDF4

xarray tutorial: https://renkulab.io/projects/gregorl/tutorial-the-xarray-ecosystem

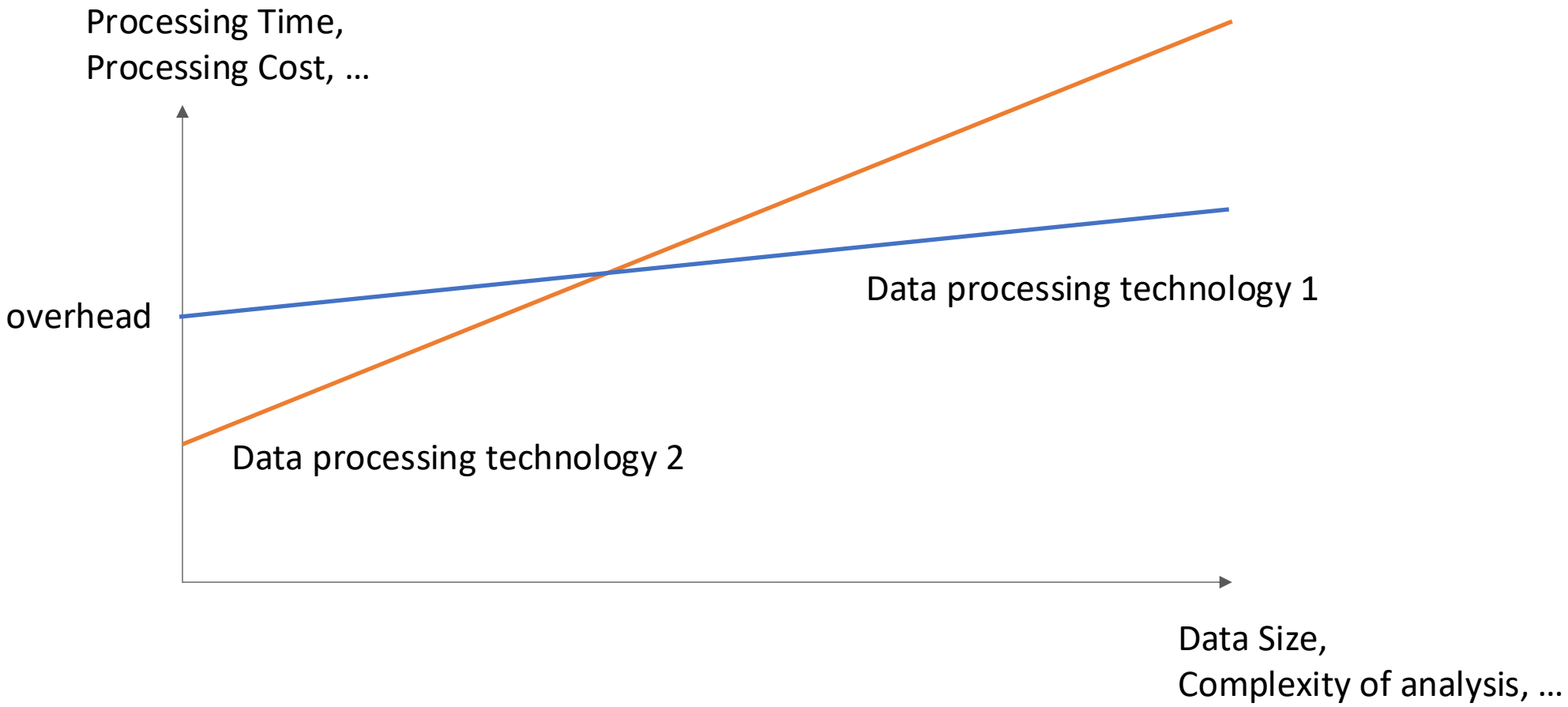
Distributed Data Processing Engines

Tool	Key Features	Typically Used With
Dask	Parallel computing on multi-servers (cluster), integrates with Pandas (Modin), Xarray, Optuna, and Scikit-Learn	Parquet, Avro, CSV, HDF5
Ray	Distributed ML and AI workloads, integrates with Modin and Scikit-Learn	Parquet, CSV, JSON
Spark	Massive scalability, SQL support, MLlib.	Parquet, ORC, Avro, CSV, JSON

Data Interchange Libraries

Tool	Key Features	Integrate with	Typically Used With
PyArrow	Foundational in-memory (zero-copy) data exchange based on Arrow IPC, Parquet I/O	Pandas, Polars, Modin, Dask, Spark	Parquet, Feather, CSV, …
PetaStorm	Data access library developed by Uber. Parallel data loading for Deep Learning, ML frameworks.	TensorFlow, PyTorch, PySpark, Spark	Parquet

Data Processing Technology – Tradeoffs



References

- Modin https://modin.org/
- VAEX https://vaex.io/
- Dask https://www.dask.org/
- Polars https://pola.rs/
- Ray https://www.ray.io/
- Duckdb: https://duckdb.org/
- Xarray: https://xarray.dev/

- Arrow: https://arrow.apache.org/docs/python/index.html

Today's check list – key objectives

- **Most of you have formed the groups**
  - Otherwise contact us
- **You have access to the exercises of module 1b**
  - You can login and clone https://dslabgit.datascience.ch/course/2025/module-1b
- **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

Start your engines

Bootstrapping into Jupyter notebooks

Jupyter Lab – Exercises module 1b

1. Start a new terminal session
2. Open a terminal and in the terminal, type:

```
git clone git@dslabgit.datascience.ch:course/2025/module-1b.git
```

3. Press enter
4. You should have a new folder

```
./module-1b
```

5. Get the data (one time)

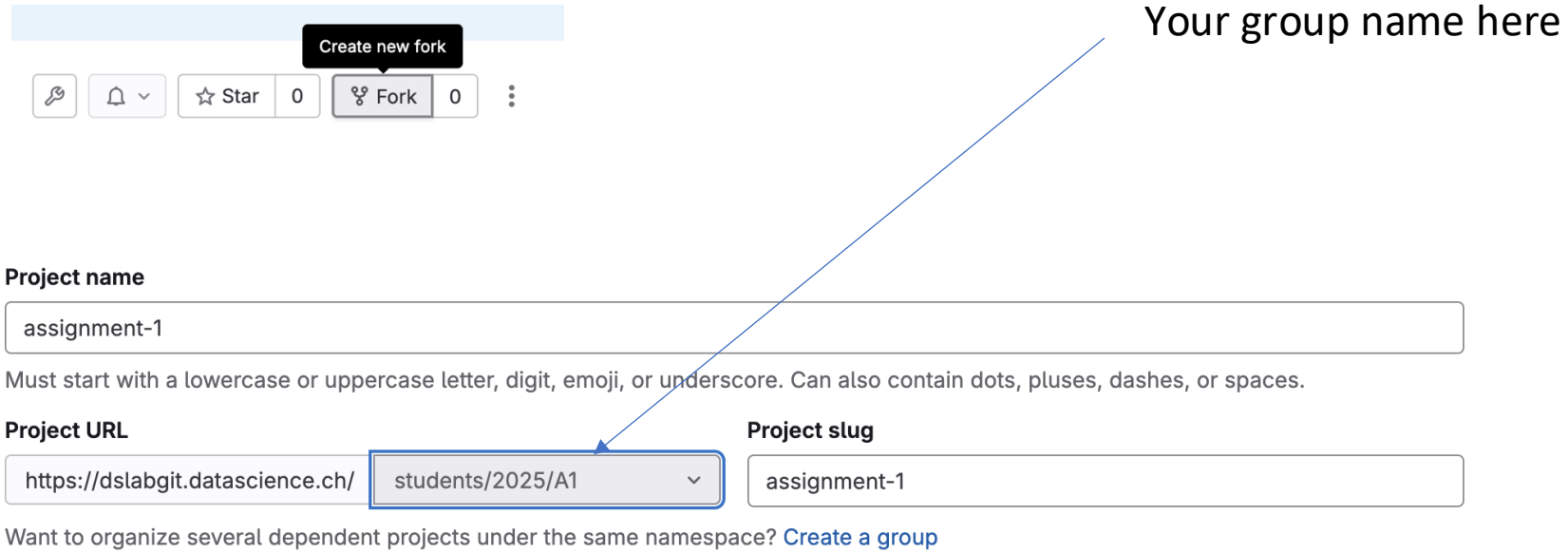
cd `./module-1b`

```
./setup.sh
```

Assignment 1

1. In Gitlab create a fork of the assignment project under your group name (/students/2025/**GroupName**) of project https://dslabgit.datascience.ch/course/2025/assignment-1

   (See appendix 2.)



Assignment 1 - continued

1. Start a new terminal session
2. Open a terminal and in the terminal, type (replace GroupName by your group name, e.g. A1)

   ```
   git clone git@dslabgit.datascience.ch:students/2025/GroupName/assignment-1.git
   ```

3. Press enter
4. You should have a new folder

   ```
   ./assignment-1
   ```

5. Get the data (one time)

   cd `./assignment-1`

   ```
   ./setup.sh
   ```

6. Open the notebook assignment-1.py

Appendix 1

GIT SETUP

Data Storage Formats - Comparison

FORMAT	BEST FOR	SCHEMA EVOLUTION	STORAGE EFFICIENCY	READ PERFORMANCE	WRITE PEFORMANCE
CSV	Simple tabular, small data (few Mb), interoperability	No (rigid)	Poor (text heavy, large)	Fast for small data	Fast for simple write
JSON	Semi-structured data, flexible schema	Yes (dynamic schema)	Poor	Slower (text parsing)	Fast for simple write
AVRO	Data pipelines, schema evolution, serialization	Yes (backward, forward)	Compact	Fast	Fast
ORC	Big data analytics (columnar, OLAP), best for Hive, Hadoop. Complex types, better suited for ACID transactions.	Yes, but limited	Highly efficient, best compression	Very fast (columnar, optimized predicate pushdown)	Moderate
PARQUET	Big data analytics, ML, (columnar, OLAP), complex types, polyvalent.	Yes (optional field)	Highly efficient	Very fast (columnar)	Moderate
HDF5 NetCDF(4)	Scientific data, large multidimensional arrays	Limited	Efficient	Fast (optimized for arrays)	Fast

Data Storage Formats – PARQUET vs ORC

FEATURE	ORC	PARQUET
File Format Type	Columnar, optimized for analytical workloads (OLAP)	Columnar, optimized for analytical workloads
ACID Transaction, Insert/Update/Delete Support	Provides native support for ACID insert/update/delete operations, with built-in compaction for delta files and updates	No native support, requires external frameworks (e.g. Delta Lake, Hudi)
Column-level indexing	Yes, with min/max indexes for predicate pushdown	Limited, no built-in column-level indexing
Compression	Optimized for efficient read with various compression formats	Supports various compression formats (e.g., Snappy, Gzip)
Performance (I/O & queries)	Faster for **write-heavy** workloads (ACID)	Better for **read-heavy** analytical queries
Use Cases	Best for Hadoop ecosystems (Hive, transactional systems)	Best for Big Data analytics (Spark, Impala) and when interoperability is a must.

Appendix 2

GIT SETUP

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]
```

Gitlab Setup

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

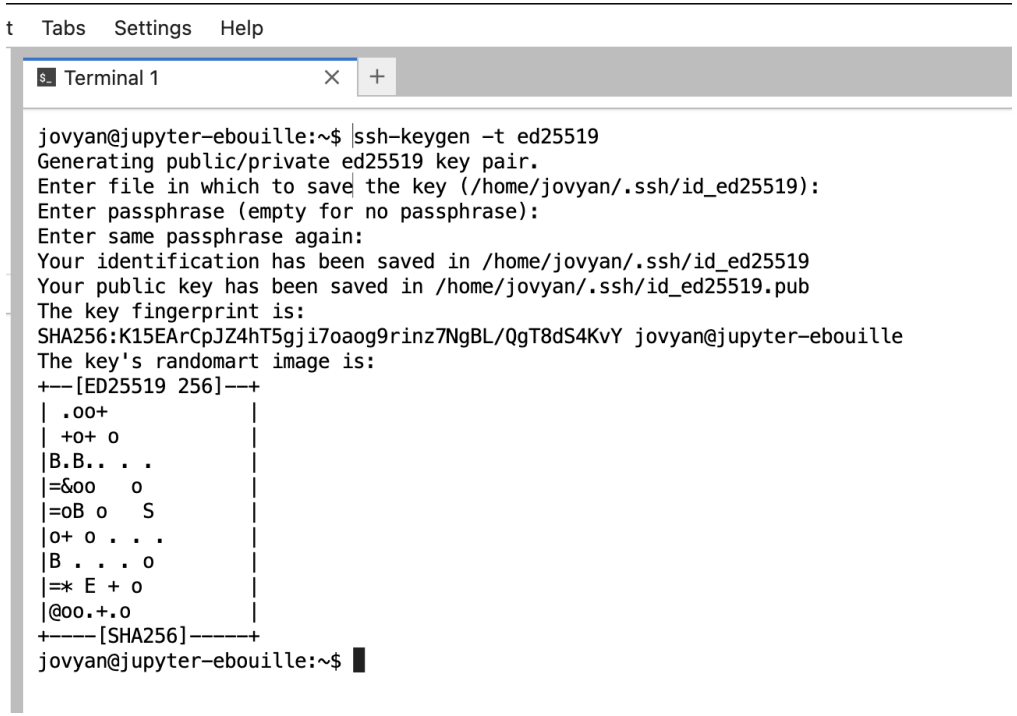
- This should be done automatically the first time you log in Jupyter Hub
- You should have received an email about a new key (com490) being added to your gitlab profile
- There should be a private ssh key in your home folder: `ls ~/.ssh/com490_key`
- An ssh-agent (key chain) should be running under 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

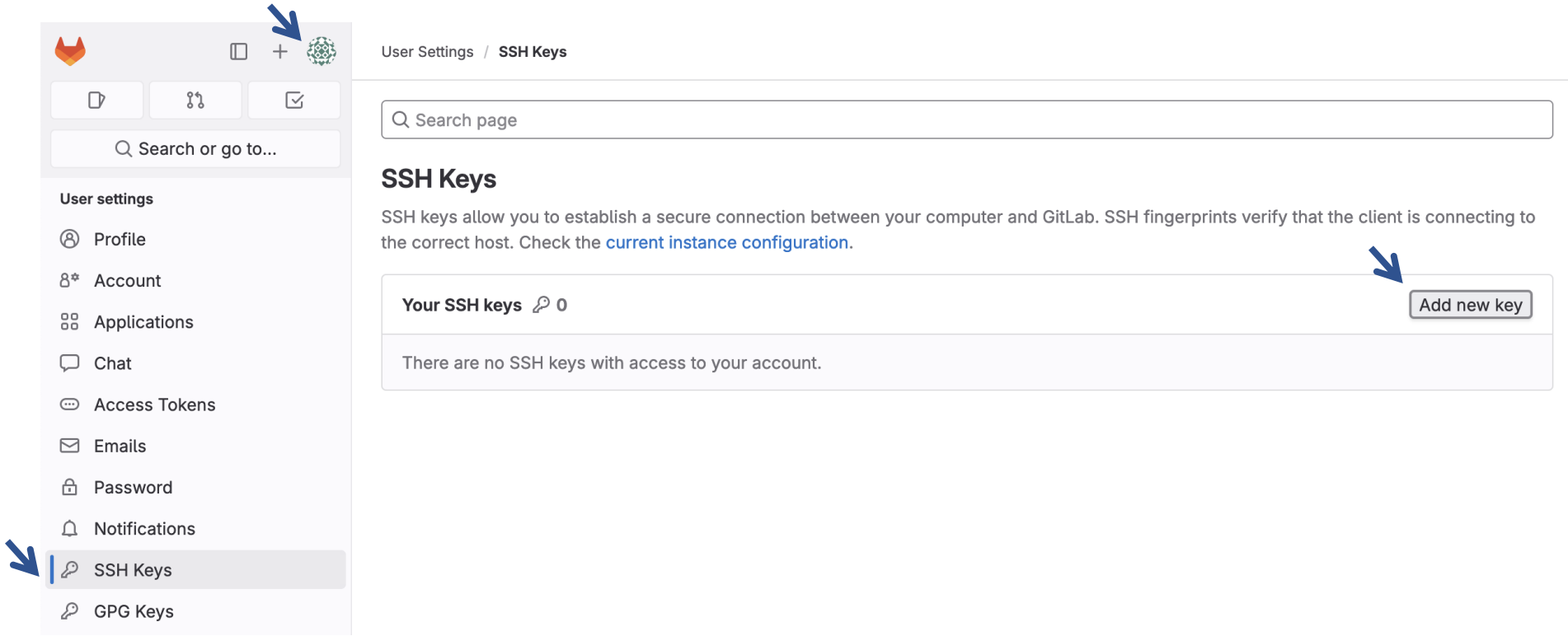
1. In a terminal enter the command to display the public key (.pub): `cat ~/.ssh/id_ed25519.pub`
2. Select and copy the content of the public key

Gitlab Setup

1. Sign in to gitlab

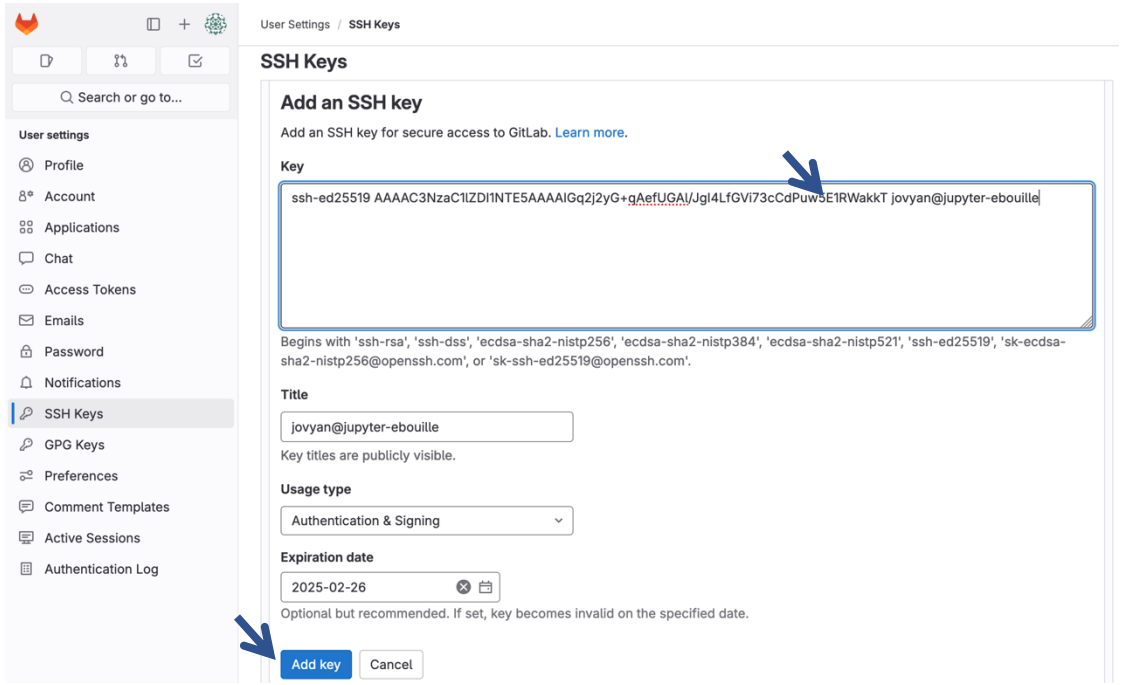
   You should have received an invitation email to set up your password, if expired you can request a new invitation.

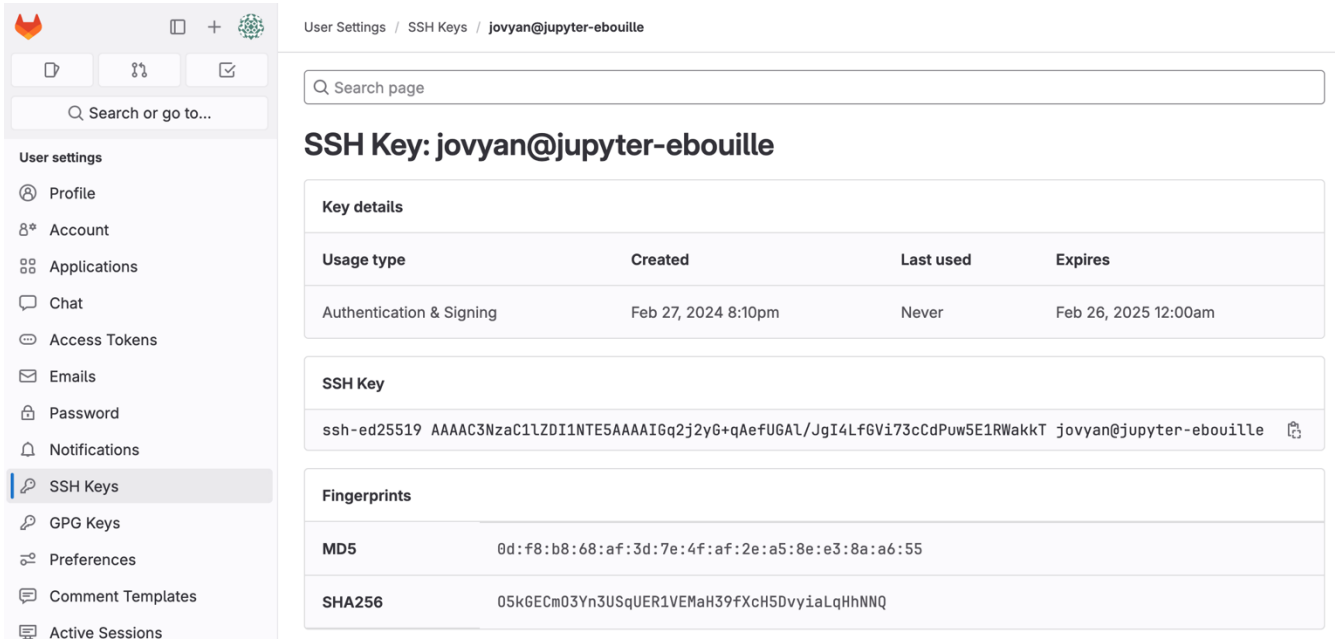
2. In your profile (click on the avatar), select '**SSH keys**', and '**Add new key**'



Gitlab Setup

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

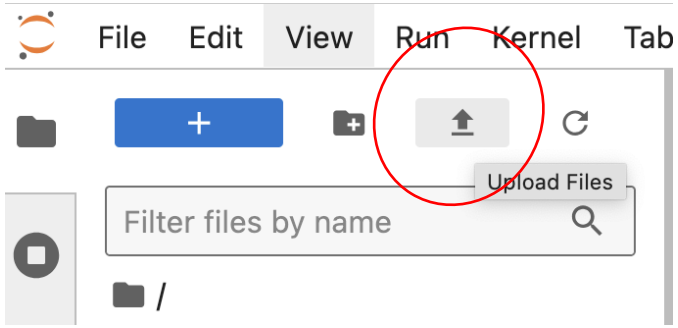




Gitlab Setup

In the same folder ~/.ssh

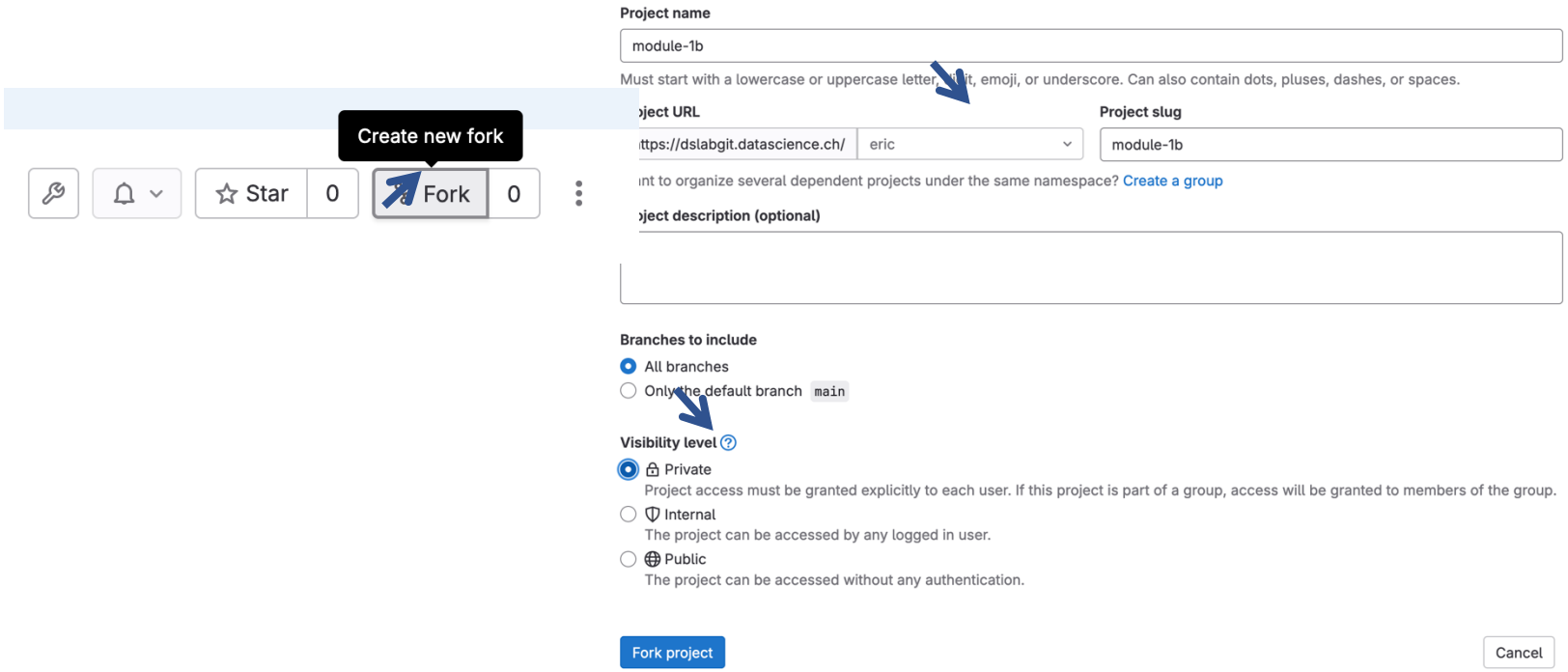
- Create or edit the file ~/.ssh/config
  - Or if this is easier create the file locally and upload it in Jupyter Hub.
- Add the following lines to it (id_ed25519 is the private key) and save




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

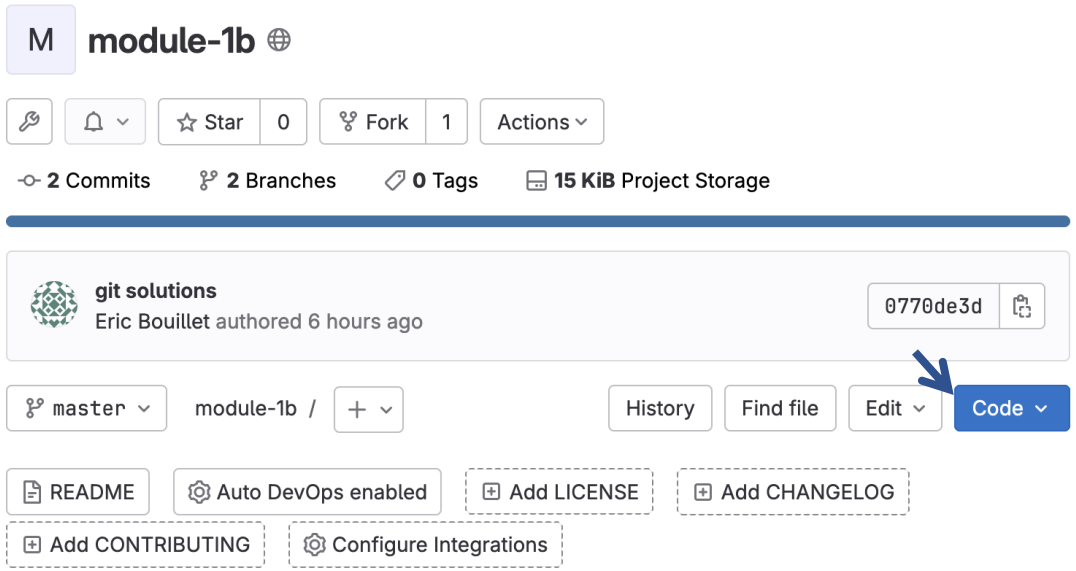
Gitlab – Making a copy (fork) of a git repository

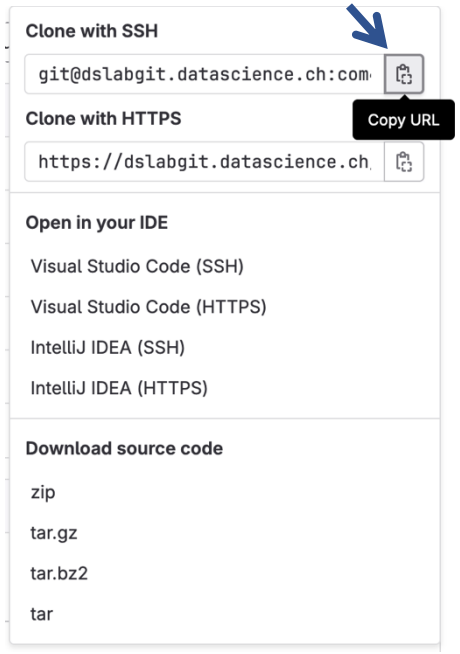
1. Sign in to gitlab and navigate to the project you want to copy
   E.g. https://dslabgit.datascience.ch/course/2025/module-1b
2. Fork the project, under your name or gitlab group name (e.g. /students/2025/A1), set the visibility to private



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*