# Recommender Systems

## Snellius Practical Guide

Compute, Storage, and SLURM Basics

Kidist Amde Mekonnen  Dominykas Seputis

k.a.mekonnen@uva.nl  d.seputis@uva.nl

University of Amsterdam

June 1, 2026

# What You Should Be Able to Do

By the end of this session, you should be able to:

- Log in to **Snellius** and run basic account checks.
- Choose the right storage location for code, data, logs, and results.
- Set up and reuse a Python environment.
- Submit, monitor, debug, and cancel jobs with **SLURM**.
- Use the shared course budget responsibly.

**Shared budget context.** The course budget is shared across **81 students** and **12 TAs**, which corresponds to roughly **30.3K SBUs per active course user**. Use it responsibly: test small, request only what you need, and cancel jobs you no longer need.

**A failed 10-minute test is fine. A forgotten 24-hour job wastes the shared budget.**

# Responsible Snellius Use

**Snellius is a shared course resource**

- We **monitor usage** throughout the course.
- Be responsible: do not block other students' work.
- Submit **one job at a time** unless you have a clear reason to do otherwise.
- Start with short test jobs before launching long runs.
- Use only the resources you need: reasonable time, memory, GPUs, and storage.

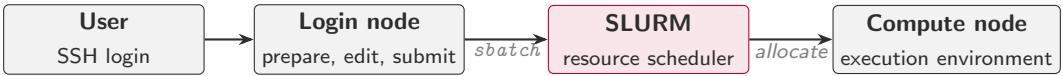
## Need help?

If you encounter any issue with Snellius, reach out to **Dominykas**, **Kidist**, or your supervising TA.

**Be fair. Test first. Do not waste shared resources.**

# Mental Model: Scheduled Computing

On a laptop, code is executed directly on the available local hardware. On Snellius, computation is mediated by a **job scheduler**: you specify the resources your experiment needs, and SLURM allocates those resources when they become available.




## Key idea

You do not run experiments directly on Snellius login nodes. You prepare a job, submit it to SLURM, and SLURM runs it on a compute node.

# Login Nodes and Compute Nodes

Snellius separates **coordination** from **execution**. This distinction is important for both system stability and fair resource use.

**Login node: coordination layer**

- edit scripts and configs;
- inspect files and logs;
- submit and monitor jobs;
- transfer small files;
- run only lightweight commands.

**Compute node: execution layer**

- train models;
- run GPU experiments;
- execute evaluations;
- perform heavy preprocessing;
- run long Python scripts.

## Operational rule

**Do not run training or heavy preprocessing on the login node.**

Use the login node only to prepare and submit jobs. All compute-intensive work must run through SLURM on compute nodes.

# Logging In to Snellius

First, generate a private and public SSH key pair, following this tutorial.

After, use the Snellius username you received from SURF, usually of the form `scurXXXX`.

**From macOS/Linux terminal or Windows PowerShell/MobaXterm:**

```
ssh -i '/path/to/keyfile' <your_scur_username>@snellius.surf.nl
```

**Example:**

```
ssh -i '~/.ssh/snellius_key' scurXXXX@snellius.surf.nl
```

After logging in, you should see a Snellius welcome message and a prompt like:

```
[scur1187@int5 ~]$
```

**If login fails**

Copy the exact error message and contact the TA.

# First Checks After Login

After logging in, run a few simple checks:

```
whoami # your Snellius username, e.g., scur1187
hostname # the login node, e.g., int5
pwd # your current directory, usually /home/<user>
groups # groups linked to your login
```

**Optional: check the shared course budget**

```
accinfo
```

**How to read `accinfo`**

`accinfo` shows the shared RecSys course account, e.g. gpuuva080. It is expected that the account contact is the course organizer. This is **not** an individual student budget.

# Where Should Files Go?

**Use the right storage location.**

Wrong storage choices cause quota issues, missing files, and wasted time.

| Location | Use for | Rule |
|---|---|---|
| `/home/$USER` | code, configs, small scripts | **Do not store datasets or checkpoints here.** |
| `/projects/prjs2120` | shared datasets, group outputs, final results | **Use this for important course files.** |
| `/scratch-shared/$USER` | temporary files and intermediate outputs | **Temporary: auto-cleaned after 14 days.** |

**Final rule: Project space** = keep it. **Scratch** = temporary. **Home** = code only.

# Storage Rules for the Course

- Use `/home/$USER` for small files only: scripts, configs, notes, and environment files.
- Use the official course project space, `/projects/prjs2120`, for: shared datasets, group folders, important logs, final results, and selected checkpoints.
- Use your assigned group folder only, e.g. `/projects/prjs2120/groups/group_01`.
- Use `/scratch-shared/$USER` only for: temporary files, intermediate outputs, and large files that can be regenerated.
- Do not store large datasets or model checkpoints in your home directory.
- Do not rely on scratch for anything you need later.

## Common failure mode

Students save final checkpoints to `/scratch-shared/`, then find that the files were automatically removed. Move important outputs to your group folder under `/projects/prjs2120/groups/` as soon as the job finishes.

# Recommended Project Structure

Course project space: `/projects/prjs2120/`

```
/projects/prjs2120/
|-- datasets/              # shared - do NOT duplicate
|-- starter_code/          # optional starter code
|-- baselines/             # reference results
|-- groups/
|   |-- group_01/
|   |   |-- code/
|   |   |-- configs/
|   |   |-- logs/
|   |   |-- checkpoints/
|   |   `-- results/
|   |-- group_02/
|   `-- group_XX/
`-- docs/
```

**Rules.** Use only your assigned group folder under `groups/`. Datasets go in `datasets/` once. Store logs, results, and selected checkpoints in your group folder. Do not store important files in scratch.

# Getting Code from GitHub

Recommended workflow: keep your code on **GitHub** and pull it directly on Snellius.

```
# log in to Snellius
ssh <user>@snellius.surf.nl

# clone your repo once
mkdir -p ~/code
cd ~/code
git clone https://github.com/<username>/<repo>.git
cd <repo>

# later: update code from GitHub
git pull
```

## Use the right tool

**GitHub**: code, scripts, configs, small text files.
`scp`/`rsync`: results, logs, checkpoints, large files, or data.

**Do not commit large datasets, checkpoints, or private credentials to GitHub.**

# Moving Large Files In and Out

When dealing with larger files, like datasets or training artifacts, git is not the best tool to use. Instead, use scp command to copy files to and from Snellius.

Run these commands from your **local terminal**, not from inside Snellius.

**Laptop → Snellius:**

```
scp my_script.py <user>@snellius.surf.nl:~/code/
scp -r my_project/ <user>@snellius.surf.nl:~/code/
```

**Snellius → laptop** *(e.g., download results):*

```
scp -r <user>@snellius.surf.nl:/projects/prjs2120/groups/group_01/results ./
```

**Sync code during development** *(incremental copy):*

```
rsync -avz --exclude '__pycache__' --exclude '.git' \
     my_project/ <user>@snellius.surf.nl:~/code/my_project/
```

# Which Transfer Method Should You Use?

## Practical note

Use `scp` for simple one-time transfers, such as copying a script, a small folder, or downloading final results.

Use `rsync` when repeatedly syncing a project folder, because it only copies files that changed.

## Project-space note

The official course project path is `/projects/prjs2120`. Use your assigned group folder under `/projects/prjs2120/groups/` for logs, results, and selected checkpoints.

## Alternative workflow

If you prefer editing files directly on Snellius, use VS Code Remote-SSH. That is covered next.

# Optional: Editing with VS Code Remote-SSH

**VS Code Remote-SSH** lets you edit files on Snellius directly from your laptop. This is useful for writing scripts, editing configs, and inspecting logs.

**Typical workflow:**

1. Install the **Remote-SSH** extension in VS Code.
2. Select commands pannel via `cmd+shift+p` and enter: `Remote-SSH: Connect to Host`
3. Select `Configure SSH Host...` option
4. Select default SSH config file *home/.ssh/config*
5. Add new host section:

```
Host snellius
  HostName snellius.surf.nl
  User <your_username>
  IdentityFile <path_to_your_private_ssh_key>
```

6. Open your code folder:

```
~/code/my_project
```

7. Edit scripts and configs in VS Code.
8. Submit experiments with `sbatch`.

# VS Code: Important Safety Rule

VS Code Remote-SSH connects to a **login node**. This is fine for editing, but not for computation.

## Use VS Code for

- editing Python scripts;
- editing YAML/JSON config files;
- reading logs;
- organizing files.

## Do not use VS Code for

- training models;
- running evaluations;
- heavy preprocessing;
- long Python scripts on the login node.

**Use VS Code to edit. Use SLURM to run. Close unused VS Code sessions.**

# Python Environment: Why It Matters

Use a dedicated Python environment for your project. This keeps package versions stable and makes your experiments easier to reproduce.

## Goal

Create one environment for your project, reuse it in all experiments, and keep the environment specification with your GitHub repository.

## Recommended workflow

Clone the code from GitHub first, then create the environment from `environment.yaml` using `conda` or `mamba`, then use the same environment inside every SLURM job script.

**Same code + same environment + same config = reproducible run.**

# Step 1: Check Whether Conda or Mamba Is Available

First check whether Conda or Mamba is already available:

```
conda --version
mamba --version
```

If one of these prints a version number, you can go directly to creating the environment.

If Conda is not available:

```
conda: command not found
```

then Conda is not available in your current shell.

**Next step**

Install Miniforge once. Miniforge is a lightweight Conda distribution based on `conda-forge`.

# Step 2: Install Miniforge if Needed

Install Miniforge once in your home directory:

```
mkdir -p ~/software
cd ~/software

wget https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-Linux-
    x86_64.sh

bash Miniforge3-Linux-x86_64.sh -b -p ~/software/miniforge3
```

Activate Conda:

```
source ~/software/miniforge3/etc/profile.d/conda.sh
conda --version
```

**Important**

Install Miniforge only once. Do not reinstall it for every experiment. If installation fails, copy the error message and ask your supervising TA.

# Step 3: Install Dependencies

After cloning the GitHub repository, create the environment from `environment.yaml`.

```
# if you installed Miniforge yourself
source ~/software/miniforge3/etc/profile.d/conda.sh

# go to your project folder
cd ~/code/<repo>

# option 1: create with conda
conda env create -f environment.yaml

# option 2: create with mamba, usually faster
mamba env create -f environment.yaml

# activate the environment
conda activate <env-name>
```

## Important

Create the environment only once. For later runs, only use `conda activate <env-name>`. Even if you create it with `mamba`, activation is still usually done with `conda activate`.

**Do not reinstall packages after every failed run. Fix the environment file.**

# Step 4: Use the Environment in Job Scripts

Add the same activation commands to every SLURM job script.

If you installed Miniforge yourself:

```
source ~/software/miniforge3/etc/profile.d/conda.sh
conda activate <env-name>
```

Then run your script:

```
cd ~/code/<repo>
python train.py --config configs/baseline.yaml
```

## Common mistake

Students often create an environment interactively, but forget to activate it inside the SLURM job script. The job then fails with errors such as `No module named torch`.

# Step 5: Modern Alternative to Conda

Conda is still widely used in scientific computing, but modern Python tools such as uv can be faster and simpler for some workflows.

Many modern Python projects now use uv because it is fast and lightweight.

1. Modify your ‘ /.bashrc‘ file to automaticly load necessary modules by adding these new lines:

```
module load 2025
module load Python/3.13.5-GCCcore-14.3.0
module load CUDA/12.9.1
```

You’ll automatically load the necessary components needed training models using GPUs each time you’ll run ‘sbatch‘ scripts. 2. Install uv:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

3. Learn how to use the tool, following the official docs.

## Example

Adding dependencies: `uv add torch`
Running python script: `uv run script.py`

# What SLURM Needs to Know

SLURM is the **job scheduler** on Snellius. It decides when and where your experiment runs.

A SLURM job script is a bash script with special `#SBATCH` directives at the top. These directives describe the resources your job needs.

**Six things SLURM needs:**

1. partition, e.g. `staging`, `gpu_mig`, `gpu_a100`, `gpu_h100`
2. number of GPUs
3. number of CPU cores
4. memory requirement, if needed
5. maximum runtime, i.e. walltime
6. command to execute

**Three commands you will use often:**

`sbatch job.sh` – submit a job
`squeue -u $USER` – list your jobs
`scancel <jobid>` – cancel a job

## How it works

You specify the resources your job needs. SLURM allocates a suitable compute node, runs your script, writes the output logs, and releases the resources when the job finishes.

# The Canonical Job Script

```
#!/bin/bash
#SBATCH --job-name=recsys_train
#SBATCH --partition=gpu_mig
#SBATCH --gpus=1
#SBATCH --cpus-per-task=9
#SBATCH --time=00:30:00
#SBATCH --output=logs/%x-%j.out
#SBATCH --error=logs/%x-%j.err

# --- environment ---
source ~/software/miniforge3/etc/profile.d/conda.sh
conda activate <your_env_name>

# --- diagnostics ---
echo "Job ID: $SLURM_JOB_ID"
echo "Running on: $SLURMD_NODENAME"
nvidia-smi
date

# --- run experiment ---
cd ~/code/my_project
python train.py --config configs/baseline.yaml

date
echo "Done."
```

Before submitting for the first time: `mkdir -p logs`
Save it with a descriptive file name, for example `recsys_train.sh`. & Submit with: `sbatch recsys_train.sh`

# What You Should Edit

For each experiment, update only the parts that need to change:

- `--job-name`: short name for the experiment.
- `--partition`: use **staging** for CPU-only jobs, **gpu_mig** for small GPU tests, and **gpu_a100** for real GPU runs.
- `--time`: maximum runtime. Start small.
- `conda activate`: your environment name.
- `cd`: path to your project code.
- `python train.py ...`: the actual command to run.

## Rule

Run a short test job first. Only submit long GPU jobs after the script, environment, data paths, and output folders are verified.

# Choosing the Right Partition

| Partition | Best use case | Smallest allocation | Cost |
|---|---|---|---|
| `staging` | CPU-only preprocessing, indexing, evaluation, data preparation | 1 CPU core + 7 GiB RAM | 2 SBU/hr |
| `gpu_mig` | **Default for GPU testing**, small models, prototyping | 1 GPU (MIG) + 9 CPU cores, 60 GiB RAM | **64 SBU/hr** |
| `gpu_a100` | Full training runs, real experiments | 1 full A100 + 18 CPU cores, 120 GiB RAM | 128 SBU/hr |
| `gpu_h100` | Only if really needed: large models, fast iteration | 1 H100 + 16 CPU cores, 180 GiB host RAM | 192 SBU/hr |

**Rule of thumb. Use the smallest resource that works.** For CPU-only work use `staging`; for small GPU tests use `gpu_mig`.

# Interactive Sessions (For Debugging)

Do not run training on the login node. If you need a GPU shell for debugging, use `srun`:

```
srun --partition=gpu_mig --gpus=1 --ntasks=1 \
     --cpus-per-task=9 --time=00:30:00 \
     --pty bash -i
```

This gives you a shell on a compute node with one GPU MIG instance for up to 30 minutes.

**Once you are in:**

```
source ~/software/miniforge3/etc/profile.d/conda.sh
conda activate <your_env_name>

python -c "import torch; print(torch.cuda.is_available(), torch.cuda.get_device_name(0))"
```

**Important**

**You are billed while the interactive allocation is active, up to the requested time.** Exit with `exit` when done, or cancel with `scancel <jobid>`.

# Estimating SBU Cost

**Cost formula**

$$\text{SBUs} = \text{allocated units} \times \text{hours requested} \times \text{SBU rate}$$

**Example:** requesting 1 A100 for 8 hours costs

$$1 \times 8 \times 128 = 1{,}024 \text{ SBUs.}$$

## Shared course budget

The course account has a shared budget of ≈**3M SBUs**. This budget is used by all student groups, TAs, and for reproducing or grading your work.

**Be a good cluster citizen:** test on `gpu_mig`, request only the time you need, and avoid leaving jobs running unnecessarily.

Course account: `gpuuva080`. Budget expires: `2026-07-10`.

# Monitoring and Managing Your Jobs

**What is running?**

```
squeue -u $USER
squeue -u $USER -o "%.18i %.12P %.25j %.8T %.10M %.10l %R"
```

**Why is my job pending?**

```
squeue -u $USER --start
scontrol show job <jobid>
```

**Watch live output**

```
tail -f logs/recsys_train-95234.out
```

**Cancel jobs**

```
scancel <jobid>
scancel -u $USER
```

## Useful job states

`PD` = pending  `R` = running  `CG` = completing  `F` = failed  `CD` = completed

# Before You Submit a Long Job

A long job that fails because of a typo wastes GPU time. Always do a cheap test first.

1. Run a 10-minute test on `gpu_mig` with a tiny subset of data.
2. Check that the SLURM `.out` log shows progress.
3. Confirm that checkpoints are saved where you expect.
4. Confirm that the test job exits cleanly.
5. **Only then** scale up the partition, GPU count, and time limit.

Cheap diagnostics to put in every job script

```
echo "Job ID: $SLURM_JOB_ID"
echo "Running on: $SLURMD_NODENAME"
nvidia-smi
python -c "import torch; print(torch.cuda.is_available())"
```

# When Things Break: Errors → Fixes

| Symptom | Most likely fix |
|---|---|
| `Permission denied on ssh` | Account not activated; reset password in SURF portal. |
| `sbatch: Batch script invalid` | Check `#SBATCH` lines for typos or unsupported flags. |
| `Unable to allocate resources` | Partition is full or request is too large; wait or try `gpu_mig`. |
| Job pending for a long time | Run `scontrol show job <jobid>` and check `Reason`. |
| `CUDA out of memory` | Reduce batch size, use gradient accumulation, or use a larger GPU. |
| `No module named torch` | Environment was not activated inside the job script. |
| `cuda.is_available() = False` | GPU was not requested, or the job is on the wrong partition. |
| Job runs but no output appears | Check `#SBATCH --output=...` and the output directory. |

# Be Responsible: Usage Is Visible

The course GPU budget is shared. Use it carefully and keep your jobs course-related.

## Important

We can monitor usage on the course account, including per-user usage, submitted jobs, partitions, and runtimes. This is mainly to manage the shared budget fairly and make sure resources remain available for everyone.

**Good practice:**

- Use `staging` for CPU-only work and `gpu_mig` for GPU tests before scaling up.
- Use `gpu_a100` or `gpu_h100` only when needed.
- Request realistic time limits.
- Cancel unused or stuck jobs.
- Keep the course account for course-related work only.

**Check budget remaining:** `accinfo` · `budget-overview`

## Tip - monitor the GPU load

To check if you are fully utilizing the selected GPU resources you can always ssh into the compute node and run `nvidia-smi` command to track the usage.

# Experiment Hygiene

Every run that might appear in your report should be reproducible.

**Save the setup**

- config file
- random seed
- git commit hash
- dataset version
- SLURM job script

**Save the evidence**

- full `.out` log
- final metrics
- best checkpoint
- training curves
- error message, if failed

## Hard rule

**Only report results that you can rerun and verify.**

A run with no config, no seed, and no log is not reliable evidence.

Use `wandb`, `tensorboard`, or simple JSON/CSV logs. Store outputs in `/projects/`, not temporary scratch.

# Top Mistakes and Better Defaults

| Mistake | Better default |
|---|---|
| Training on the login node | Submit with `sbatch`; use `srun --pty bash` only for short debugging. |
| Requesting many GPUs "just in case" | Start with one GPU. Scale up only if your code can use multiple GPUs. |
| Submitting a long job without testing | Run a short `gpu_mig` test first. |
| Putting datasets in `$HOME` | Use the shared project dataset directory and symlink if needed. |
| Saving final results to scratch | Move important outputs to project space. Scratch is temporary. |
| Creating many duplicate Conda environments | Keep one clean environment per project; remove unused environments. |
| Re-submitting failed jobs blindly | Read the `.out`/`.err` log and fix the actual error. |
| Leaving duplicate jobs running | Check `squeue -u $USER`; cancel duplicates with `scancel`. |

# Cheat Sheet: Basic Commands

**Login and orient**

```
ssh <user>@snellius.surf.nl
accinfo
myquota
squeue -u $USER
```

**Submit / monitor / cancel**

```
sbatch job.sh
squeue -u $USER
scancel <jobid>
tail -f logs/<jobname>-<jobid>.out
```

**Environment**

```
source ~/software/miniforge3/etc/profile.d/
    conda.sh
conda activate <your_env_name>
```

**Useful in scripts**

```
echo $SLURM_JOB_ID
echo $SLURMD_NODENAME
nvidia-smi
```

**Partitions**

`staging` = CPU-only, 2 SBU/hr
`gpu_mig` = GPU debug, 64 SBU/hr
`gpu_a100` = full A100, 128 SBU/hr
`gpu_h100` = full H100, 192 SBU/hr

# Cheat Sheet: Debugging and File Transfer

**GPU interactive debugging**

```
srun --partition=gpu_mig --gpus=1 \
     --cpus-per-task=9 --time=00:30:00 \
     --pty bash -i
```

**CPU-only debugging**

```
srun --partition=staging \
     --cpus-per-task=1 --mem=7G \
     --time=00:10:00 --pty bash -i
```

**Move files**

```
# laptop -> Snellius
scp file <user>@snellius.surf.nl:~/

# Snellius -> laptop
scp <user>@snellius.surf.nl:path .

# sync folder
rsync -avz src/ <user>@snellius.surf.nl:dst/
```

**Reminder.** Use `staging` for CPU-only work and `gpu_mig` for GPU tests. Upgrade to `gpu_a100` only after the job works.

# Going Further

When you're comfortable with the basics, three things are worth learning:

- **Job arrays** (`#SBATCH --array=0-9`), run 10 hyperparameter configs in one submission.
- **Multi-GPU training** (`torchrun`, `deepspeed`, `accelerate`), for models that don't fit on one A100.
- **Mixed precision** (`torch.amp`), 2× throughput, often free for transformer training.

**Documentation worth bookmarking:**

- Snellius user guide: https://servicedesk.surf.nl/wiki/display/WIKI/Snellius
- UvA DL tutorial: https://uvadlc-notebooks.readthedocs.io/ (Guide 1)
- Partition + accounting reference: search "Snellius partitions and accounting"

## If you're stuck for >30 minutes

**Ask on the course channel before burning more time.**

Include: job ID, partition, the `.err` log, and what you've already tried.

# Take-Home

1. Snellius is **shared**. Every wasted GPU-hour consumes the shared course budget.
2. **Login nodes are for editing**, compute nodes are for compute. Never the reverse.
3. Use `staging` for CPU-only work and `gpu_mig` for GPU tests. Upgrade to `gpu_a100` only after the job works.
4. **Test small, then scale.** A 10-minute test catches 90% of failures.
5. **Save outputs to project space.** Scratch is wiped every 14 days.
6. **Track every run**, config, seed, commit, log. Reproducibility is not optional.

**Debug small. Run large. Document everything.**