BinAC 2 Training

Part 2: Working on BinAC 2

University of Tübingen · High Performance and Cloud Computing Division

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN



BinAC 2

By 17:00 you can do this

```
#!/bin/bash
#SBATCH --partition=compute
#SBATCH --cpus-per-task=8
#SBATCH --mem=16G
#SBATCH --time=02:00:00
#SBATCH --array=1-3
#SBATCH --output=logs/map_reads_%A_%a.out

module load bio/bowtie2/2.5.4
module load bio/samtools/1.24
ws=$(ws_find training)
sample="sample_${SLURM_ARRAY_TASK_ID}"

cd "$TMPDIR"
bowtie2 -p "$SLURM_CPUS_PER_TASK" -x "$ws/ref/genome" \
    -1 "$ws/fastq/${sample}_R1.fastq.gz" \
    -2 "$ws/fastq/${sample}_R2.fastq.gz" \
  | samtools sort -@ 4 -o "${sample}.bam" -

samtools index "${sample}.bam"
cp "${sample}.bam" "${sample}.bam.bai" "$ws/results/"
```

This afternoon you learn to

- Choose the right place for your data: `/pfs/10/work`, `/pfs/10/project`, `$TMPDIR`
- Load software with modules, or install it yourself with Conda: `module load`, `conda activate`
- Write job scripts, submit them, and follow them in the queue: `#SBATCH`, `sbatch`, `squeue`
- Request the resources your job needs: `--cpus-per-task`, `--mem`, `--time`
- Check how well your job used them: `seff`, `sacct`
- Make your analysis repeatable: fixed versions, logs, scripts instead of typing

Before we start

1. Log in: `ssh USERNAME@login.binac2.uni-tuebingen.de`
2. Run `hostname`: you should see `login01`, `login02` or `login03`
3. Done? Green sticky note, or raise your hand in the video call

This afternoon assumes you know: `cd`, `ls`, `cp`, `nano`, wildcards, variables

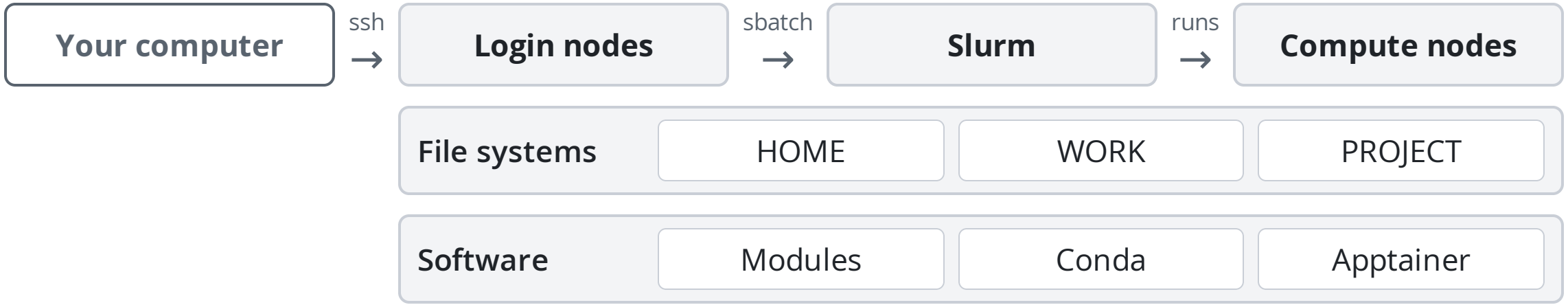
Joining only this afternoon? The morning slides are in `/pfs/10/project/training_material/slides_2026-11`

Agenda

1. Getting access
2. Login nodes
3. Where to put your data
4. Software
5. Batch jobs with Slurm
6. Efficient use of resources
7. Reproducibility

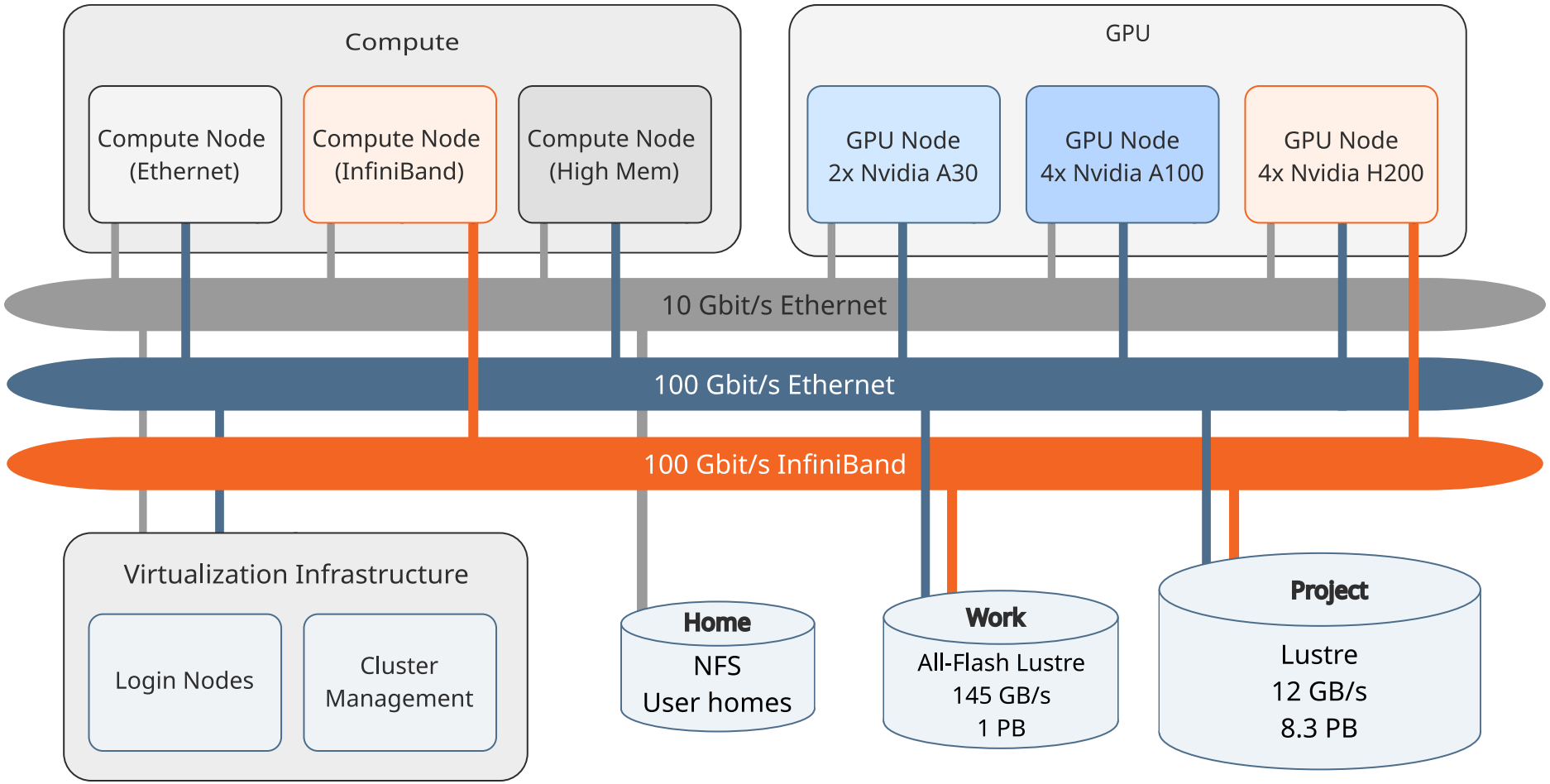
Break around 15:30

How you use BinAC 2



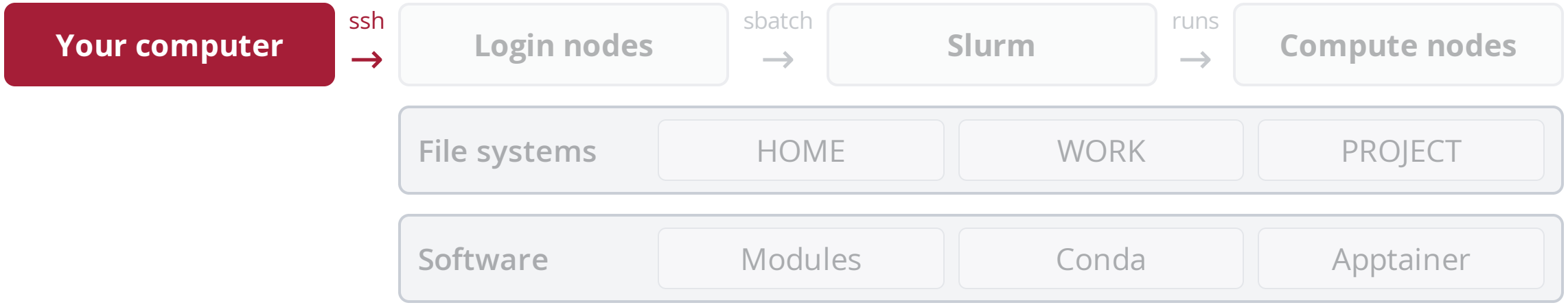
- You log in to a **login node** and prepare your work there
- With `sbatch` you hand a job to **Slurm**, which runs it on a **compute node**
- All nodes see the same **file systems** and the same **software**

BinAC 2 at a glance



Details: BinAC 2 hardware and architecture

Getting access



What you need to log in

1. Entitlement

Your home university allows you to use bwForClusters (**bwForCluster entitlement**)

Step A: Entitlement

2. Compute project

You are a member of a **compute project** (Rechenvorhaben, RV) for BinAC 2

Step B: Compute project

3. Account

You registered for BinAC 2 and set a **service password**

Step C: Registration

4. Network

You are in the **BelWü** network: on campus or via your university's **VPN**

Login

Login problems

Message	Check
`Could not resolve hostname`	typo in the host name?
`Connection timed out`	VPN connected?
`Permission denied`	Correct user name with **prefix**? **service** password? account still valid?
a message, then `Connection closed`	read the message: maintenance? see our announcements
`REMOTE HOST IDENTIFICATION HAS CHANGED`	ask us before you do anything

Still stuck? Send us the full error message and the output of `ssh -v`

Less typing: `~/.ssh/config`

On **your own computer**, in `~/.ssh/config`:

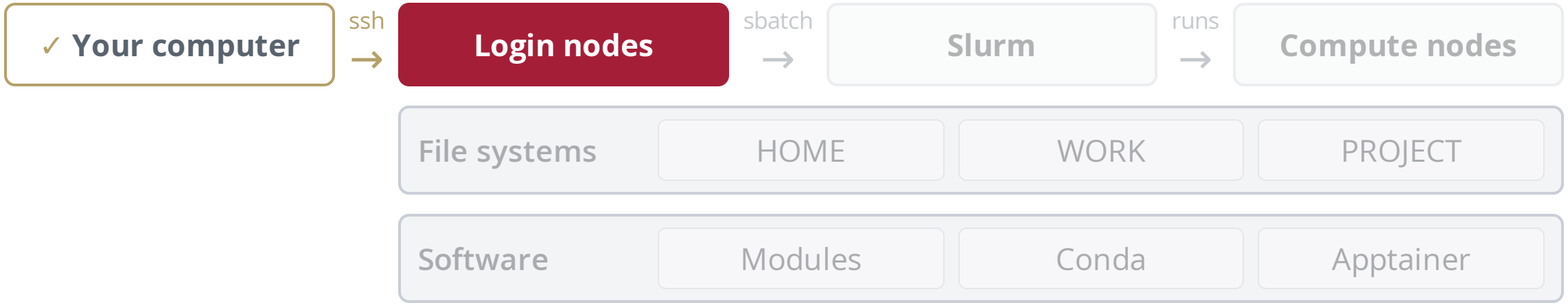
```
Host binac2
    HostName login.binac2.uni-tuebingen.de
    User tu_abc01
```

Then:

```
ssh binac2
scp results.csv binac2:training/
rsync -av data/ binac2:training/data/
```

- Works on Linux, macOS and Windows (`C:\Users\NAME\.ssh\config`)
- The name after `Host` is your choice

Login nodes


✓ Your computer
ssh
Login nodes
sbatch
Slurm
runs
Compute nodes
File systems
HOME
WORK
PROJECT
Software
Modules
Conda
Apptainer

Login nodes: what to do there

Yes

- Edit files and scripts
- Copy data to and from BinAC 2
- Compile, install software
- Submit and watch jobs

No

- Long or heavy computations: use a job
- Programs you do not know
- `squeue` in a loop every second

You share the login nodes with all other users.

Quiz: Login node or job?

Which of these should run as a **job**?

1. Unpacking a 2 GB archive you just uploaded
2. `nano job.sh`
3. Aligning 50 million reads against a genome
4. `module avail`

Three login nodes, one system

```
hostname                    # login01, login02 or login03
```

- `login.binac2.uni-tuebingen.de` sends you to **one** of the three login nodes
- **Same files** everywhere: HOME, WORK and PROJECT are shared
- **Different processes:** a program running on `login01` is not visible on `login02`
- Back to a running `screen` or `tmux` session? Choose the node with the port: `ssh -p 2221 ...` (login01), `2222` (login02), `2223` (login03)

Details: BinAC 2 login

Watch your own processes

```
sleep 300 &                  # start in the background: prompt comes back at once
top -u $USER                 # your processes, live; q to quit
ps -u $USER                  # your processes, as a list
kill PID                     # stop a process (PID from top or ps)
kill -9 PID                  # if it does not stop
```

```
    PID USER      %CPU %MEM     TIME+ COMMAND
 482113 tu_abc01  199.8  6.2   12:41.07 bowtie2-align-s
```

- `COMMAND &` starts a program in the background, you can keep typing
- `%CPU` 200: two CPUs at full load, your limit on a login node. That belongs in a job
- Started something too big by mistake? `Ctrl+C`, or `kill` it
- Limit per user on each login node: **2 CPUs, 16 GB memory**

Make your shell comfortable

Add to `~/.bashrc`:

```
HISTSIZE=50000; HISTFILESIZE=50000      # remember more commands
HISTTIMEFORMAT="%F %T  "                 # history with date and time
shopt -s histappend                      # several terminals: append, not overwrite
alias lt='ls -lht'                       # newest files first
alias sq='squeue --me'                   # your jobs (this afternoon)
```

- Activate the changes: `source ~/.bashrc` or log in again
- `type NAME` shows what a name really is: program, alias or shell built-in
- `history | grep sbatch`: how did I submit that job last week?

Exercise: Make yourself at home

Goal: look around on the login node and set up your shell

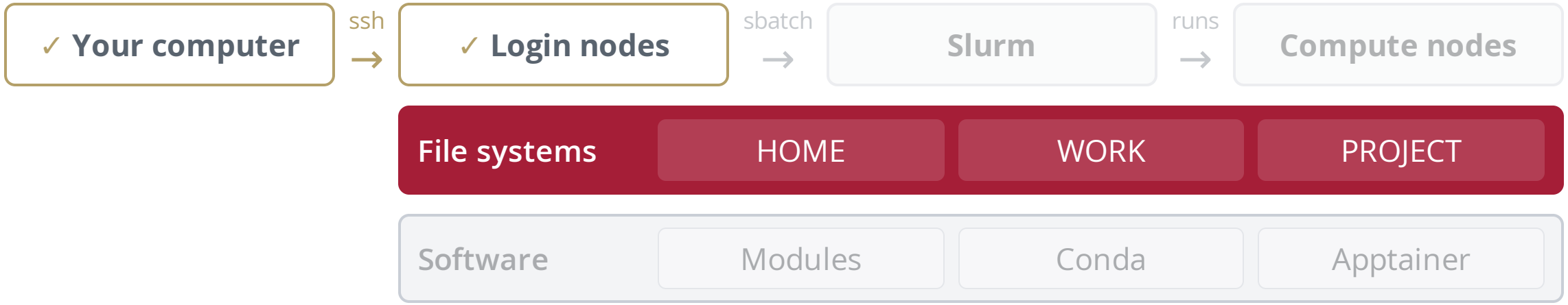
1. Find out on which login node you are
2. Start `sleep 300 &`, find it with `top -u $USER` or `ps -u $USER`, and stop it with `kill`
3. `ll` is not a program: what does it run, and where is it defined?
4. Add the history settings and an alias `lt` to your `~/.bashrc` and activate them

Expected result: `lt` works, `history` shows dates

> **Help yourself:** `type ll`, and the system wide settings are in `/etc/profile.d/`

Bonus: start `sleep 300 &` again. Can you see it from a second terminal?

Where to put your data



File systems

	Path	For	Limit	Backup	Lifetime
HOME	`$HOME`	scripts, configs	45 / 60 GB	nightly	permanent
WORK	`/pfs/10/work`	data you can recreate: intermediate files, job output	none	no	30 days, extendable
PROJECT	`/pfs/10/project`	data to keep and share: raw data, references, results	none (yet)	no	permanent
TMPDIR	`$TMPDIR`	temporary files of one job, local to the node	node disk	no	until the job ends

Details: BinAC 2 file systems

Your home directory

```
/home/tu/tu_tu/tu_abc01
├── .bashrc          ← runs at every login: aliases, variables, PATH
├── .bash_profile    ← login settings, usually just loads .bashrc
├── .ssh/            ← SSH keys and config (keep private!)
├── .ws_user.conf    ← workspace reminders
├── .condarc         ← Conda settings                          ┐
├── .conda/          ← Conda environments and packages         │
├── .config/         ← settings of many programs               │ created
├── .cache/          ← caches of pip and others                │ later, by
├── .local/          ← programs from pip install --user        │ programs
├── bin/             ← your own scripts, add to PATH           │ or by you
└── projects/        ← your job scripts and analysis code      ┘
```

- HOME is for **small, important** files: scripts, configs. It has a backup
- Change `.bashrc` with care: an error there can break every login
- Do **not** put `module load` or `conda activate` in `.bashrc`: load them in your job scripts

How full is my space?

```
my-quota                    # HOME, WORK and PROJECT usage
du -sh ~/training           # size of one directory
```

```
USER/GROUP   Filesystem                  Limit (s/h) [GB]   Usage [GB]
-------------------------------------------------------------------
tu_abc01     /home/tu/tu_tu/tu_abc01       45.0 / 60.0          12.3
tu_abc01     /pfs/10/work                   0.0 / 0.0           96.0
bw16f003     /pfs/10/project                0.0 / 0.0            3.7
-------------------------------------------------------------------
```

- **Soft limit:** you can write for 7 more days, then it acts like the hard limit
- **Hard limit:** no more writing, programs fail

`$HOME` sweet `$HOME`

Is your home directory full? Find the culprits:

```
my-quota                                # how full is it?
du -h --max-depth=1 ~ | sort -h         # size of each directory, largest last
ls -lha ~                               # all files, including hidden ones
```

```
1.4G    /home/tu/tu_tu/tu_abc01/.cache
9.2G    /home/tu/tu_tu/tu_abc01/.conda
29G     /home/tu/tu_tu/tu_abc01/results
42G     /home/tu/tu_tu/tu_abc01
```

- Usual culprits are **hidden**: `.cache`, `.conda`
- Clean up: `conda clean --all`, `rm -r ~/.cache/pip`
- Data does not belong in HOME: move it to WORK or PROJECT (next slides)
- HOME is for scripts and configs: small, but backed up

What is on WORK?

```
ls -ld /pfs/10/work
mkdir /pfs/10/work/my_data
```

```
drwxr-xr-x. 262 root root 90624 Nov 13 15:57 /pfs/10/work
mkdir: cannot create directory '/pfs/10/work/my_data': Permission denied
```

You cannot simply create a directory on WORK. How do you get space there? → next slides

Workspaces: your space on WORK

```
ws_allocate training 7        # create workspace "training" for 7 days
ws_list                       # all your workspaces, with expiry dates
ws_find training              # print its path
cd "$(ws_find training)"      # change into it
ws_extend training 30         # extend before it expires
ws_release training           # release it when you are done
```

- Up to **30 days**, extendable **5 times**; e-mail reminder one week before expiry
- Expired: restorable for 14 days, then moved away, later deleted
- By default only you can read your workspace (sharing: later in this chapter)

Workspace reminders: `~/.ws_user.conf`

This file is **already in your home directory**, with defaults set by us:

```
mail: firstname.lastname@uni-tuebingen.de   # where the reminder mails go
duration: 30                                # default lifetime of new workspaces, in days
reminder: 7                                 # first reminder this many days before expiry
```

- `ws_allocate` and `ws_extend` use these settings automatically
- You **can** change them, but it is your risk: wrong address or no reminder, no warning
- Reminder mails and expiry happen on weekdays only: extend in time, not on the last day
- An expired workspace can be restored for 14 days, then it is gone

Sharing a workspace

With your project group: everybody in the group can read and write

```
ws_allocate -G bw16f003 shared_data 30     # group workspace for project bw16f003
```

With single users: read only

```
ws_share share training tu_xyz02           # give tu_xyz02 read access
ws_share list training                     # who has access?
ws_share unshare training tu_xyz02         # take it back
```

- Group workspace: shared **working data** of a project. Data to keep goes to PROJECT
- Expired by accident? `ws_restore -l` lists what can still be restored

Details: Workspaces on the bwHPC wiki

PROJECT: shared and permanent

```
id $USER | grep -o 'bw[^)]*'      # your project ID, e.g. bw16f003
ls /pfs/10/project/bw16f003
```

- One directory per compute project, all members can read and write
- For data to keep and share: raw data, references, final results
- Talk to your project members about the structure
- Moving data between WORK and PROJECT: `rsync` or `cp`, not `mv` (see later)

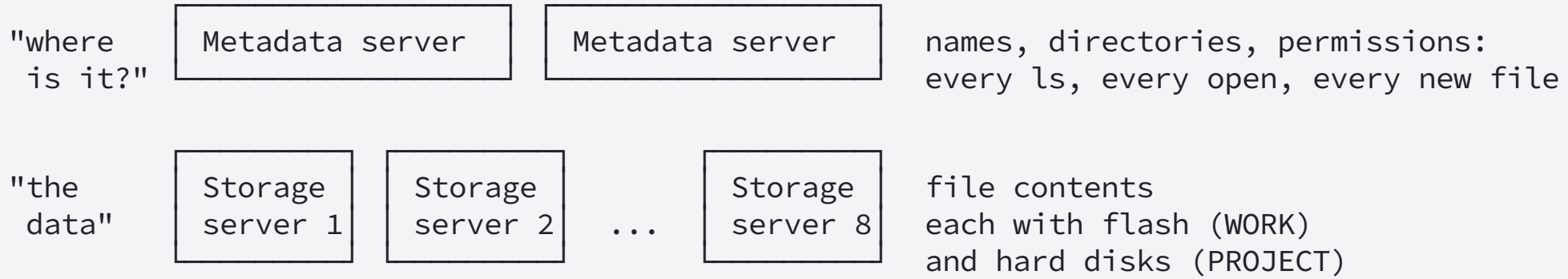
Shortcuts to your data

Symbolic links: a name in your home directory that points somewhere else

```
ln -s /pfs/10/project/bw16f003 ~/project
ln -s "$(ws_find training)" ~/work
cd ~/work                          # same as cd "$(ws_find training)"
ls -l ~                            # project -> /pfs/10/project/bw16f003
```

- The data stays where it is, only the shortcut is in HOME: no quota used
- Workspace expired? The link breaks: remove it with `rm ~/work` (removes the link, not the data)

How Lustre stores your files



- Each file lives on **one** storage server: many files and jobs spread over all of them
- Large files, read in big blocks: **fast**
- Many small files: every file costs a metadata operation: **slow for everyone**
- Changing a few bytes inside big files over and over: **slow**

Moving data from WORK to PROJECT: not with `mv`

```
mv "$(ws_find training)/results" /pfs/10/project/bw16f003/       # wrong
```

- WORK and PROJECT are **one** Lustre file system: `mv` only changes the **name**
- The data **stays on the flash disks** of WORK, now under a PROJECT path
- Flash fills up for everyone, and your data never reaches the large disks

```
rsync -av "$(ws_find training)/results/" /pfs/10/project/bw16f003/results/   # right
rm -r "$(ws_find training)/results"                                        # after checking
```

- `cp` or `rsync` writes the data **new**, onto the hard disks of PROJECT

`$TMPDIR`: the local disk of your job

```
cd "$TMPDIR"                                  # 1. go to the local disk
tar -xzf "$ws/reference.tar.gz"               # 2. unpack many small files there
my_tool --threads "$SLURM_CPUS_PER_TASK" ...  # 3. work locally: fast
cp results.bam "$ws/results/"                 # 4. copy results back
```

- A new, empty directory **for each job**, on the disk of the compute node
- Fast for many small files and lots of reading and writing: no load on Lustre
- Only visible on **this node**, during **this job**
- **Deleted when the job ends**: everything you did not copy back is gone
- The disk is **shared** with all jobs on the node and not managed by Slurm: no guaranteed space

Rules for WORK and PROJECT

- **Few large files** instead of many small ones: pack them with `tar`
- **Compress** text data: FASTQ, SAM, FASTA (`gzip`, or BAM instead of SAM), see Data compression
- **Where a job reads and writes depends on the tool**: many tools read slowly enough to work directly on PROJECT. Check the tool's page in our software documentation
- **Never** write the same file from two nodes at once
- `$TMPDIR` is wiped when the job ends: copy results back

Quiz: Where does it go?

Match each item with the best place: HOME, WORK, PROJECT or `$TMPDIR`

1. Your job scripts and configs
2. Raw FASTQ files from the sequencing facility, read by your jobs for months
3. Thousands of small temporary files a tool writes during one job
4. 300 GB of intermediate files from a large sample, deleted after the next step
5. Final results your whole group needs next year

Exercise: Your workspace

Goal: check your space and prepare a workspace for today

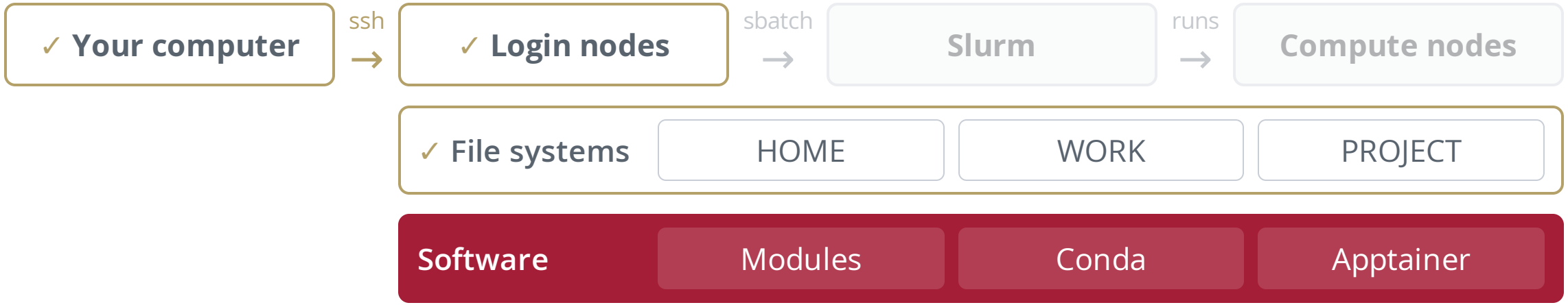
1. Check your usage with `my-quota`
2. Allocate a workspace `training` for 7 days and change into it
3. Create the directories `results` and `logs` in it
4. Look at the example data, it stays where it is: `ls -lh /pfs/10/project/training_material/data`

Expected result: `ls` in the workspace shows `results` and `logs`

> **Help yourself:** `ws_allocate --help`, the slides above

Bonus: find your project directory and look inside

Software


✓ Your computer
ssh
✓ Login nodes
sbatch
Slurm
runs
Compute nodes
✓ File systems
HOME
WORK
PROJECT
Software
Modules
Conda
Apptainer

Command not found

```
samtools --version
type samtools
echo $PATH | tr ':' '\n'      # one directory per line
```

```
bash: samtools: command not found
bash: type: samtools: not found
/home/tu/tu_tu/tu_abc01/.local/bin
/home/tu/tu_tu/tu_abc01/bin
/usr/local/bin
/usr/bin
/usr/local/sbin
/usr/sbin
```

- The shell only looks for programs in the directories listed in `$PATH`
- `samtools` is installed on BinAC 2, just not in one of these directories
- So: how do you get software into your `$PATH`?

Where software comes from

Source	Who installs	Typical use	Today
Operating system	admins	`bash`, `gzip`, `nano`, …	already in `$PATH`
Modules	admins, on request	compilers, MPI, CUDA, common tools	hands-on
Conda	you	Python, R, bioinformatics tools	hands-on
Containers (Apptainer)	you	complex software, existing images	overview

Missing something many users need? Ask us for a module.

Your environment

```
env                     # all environment variables of this shell
echo $PATH              # one of them: where the shell looks for programs
```

- **Environment:** the set of variables of **one** shell, e.g. `PATH`, `HOME`, `USER`
- Programs you start from this shell **inherit** it, and so do jobs you submit
- Every new shell starts fresh: new terminal, new login, new job
- `module load` and `conda activate` change the environment, **only of this shell**

Modules

```
module avail samtools              # which versions are installed?
module load bio/samtools/1.24      # load one
type samtools                      # now found: where?
module list                        # what is loaded?
module help bio/samtools/1.24      # documentation and examples
module purge                       # unload everything
```

- `module load` puts the program into your `$PATH`: nothing is installed or copied
- Dependencies are loaded automatically, one version of a program at a time
- Tools that produce your results: load with **version**, `bio/samtools/1.24`.
  Helpers like `devel/miniforge`: without version, you get the newest
- Example job scripts: `$SAMTOOLS_EXA_DIR`, in general `$NAME_EXA_DIR`

Exercise: Load a module

Goal: see what `module load` changes

1. Search for available `samtools` versions
2. Run `type samtools`, load `bio/samtools/1.24`, run `type samtools` again
3. Which environment variables did the module set? Search the output of `env`
4. Change into the directory with the example job scripts and look at one

Expected result: `type` finds `samtools`, `env` shows variables with `SAMTOOLS` in the name

> **Help yourself:** `module help`, `env`, `grep`, `less`

Bonus: what does `type samtools` say after `module purge`?

Quiz: What does `module load` do?

You run `module load bio/samtools/1.24`. What happens?

1. samtools is downloaded and installed into your home directory
2. samtools is copied into your current directory
3. Environment variables like `PATH` change in your current shell
4. samtools is now available for all users on BinAC 2

What is Conda?

- A **package manager**: installs programs together with all their dependencies
- Not only Python: also R, C/C++ tools, bioinformatics software
- No admin rights needed: everything goes into **your** directories
- An **environment manager**: separate sets of packages, one per project or tool

```
module load devel/miniforge      # Miniforge: conda, mamba, and conda-forge as channel
conda --version
```

- **Miniforge** is the recommended Conda distribution: open source channels only

Channels: where packages come from

Channel	Content	Use
`conda-forge`	community, general software	yes, default in Miniforge
`bioconda`	bioinformatics tools, builds on `conda-forge`	yes, add it once
`defaults`, `anaconda`	Anaconda's own packages	**no**: paid licence for universities

Add `bioconda` once, as recommended by Bioconda:

```
conda config --add channels bioconda
conda config --add channels conda-forge          # conda-forge back to first place
conda config --set channel_priority strict
conda config --show channels                     # conda-forge, bioconda
```

Where Conda puts things

```
conda info                          # look for "envs directories" and "package cache"
du -sh ~/.conda/*                   # how much space does it take?
```

```
~/.conda/
├── envs/           ← your environments: one directory each, many small files
│   └── seqtools/
└── pkgs/           ← package cache: every downloaded package, kept for reuse
```

- Both are in **HOME** by default: HOME has a quota of 45 GB
- One environment: often several GB. The cache grows with every install
- `conda clean --all` empties the cache: safe, packages are downloaded again if needed

Move Conda out of HOME

Our suggestion: a directory for you in your **project**

```
mkdir -p /pfs/10/project/bw16f003/$USER/conda
conda config --prepend envs_dirs /pfs/10/project/bw16f003/$USER/conda/envs
conda config --prepend pkgs_dirs /pfs/10/project/bw16f003/$USER/conda/pkgs
conda config --show envs_dirs pkgs_dirs
```

	HOME	WORK	PROJECT
Space	45 GB, fills up fast	limited, shared by all	**most space**
Lifetime	permanent	workspace expires	**permanent**
Good for	`environment.yml`	short tests	**environments, cache**

- A guideline, not a rule: talk to your group, maybe share environments

Create and use an environment

```
conda create -n seqtools seqkit samtools=1.24    # new environment, versions optional
conda env list                                   # all your environments
conda activate seqtools                          # use it: prompt starts with (seqtools)
seqkit version
conda install -n seqtools csvtk                  # add a package later
conda list -n seqtools                           # what is installed?
conda deactivate                                 # back to normal
```

```
(seqtools) [tu_abc01@login01 ~]$
```

- `conda activate` changes your shell environment, **only in this shell**
- Install all packages of a tool chain at once: one solve, fewer conflicts
- Installing more later works, but may change versions of packages already there

Quiz: Where is my tool?

Yesterday you created and used the environment `seqtools`. Today, in a new terminal:

```
$ seqkit version
bash: seqkit: command not found
```

What is the most likely reason?

1. The environment was deleted overnight
2. The environment is not active in this shell
3. `seqkit` is not in the `bioconda` channel any more
4. Your HOME is full

Environment files: write it down, build it again

```yaml
# seqtools.yml
name: seqtools
channels:
  - conda-forge
  - bioconda
dependencies:
  - seqkit
  - samtools=1.24
```

```
conda env create -f seqtools.yml                # build the environment from the file
conda env update -f seqtools.yml --prune        # after editing the file
conda env export --from-history > seqtools.yml  # file from an existing environment
```

- Write the file **first**, then create the environment from it: the file is the recipe
- Keep it in HOME next to your job scripts: small, backed up
- `--from-history` : only what you asked for, works on other systems too

Snapshots: record the exact versions

```
conda env export -n seqtools > seqtools.lock.yml              # every package, version and build
conda env export -n seqtools --no-builds > seqtools.lock.yml  # or: versions only, more portable
conda list -n seqtools --explicit > seqtools.spec.txt         # exact package files (linux-64 only)
```

```
conda env create -n seqtools-2026 -f seqtools.lock.yml        # rebuild from the snapshot
conda create -n seqtools-2026 --file seqtools.spec.txt        # rebuild, no solving: fast and exact
```

File	Contains	Use it for
`seqtools.yml`	what you asked for	the recipe: change and update it
`seqtools.lock.yml`	all packages with versions	a snapshot, next to your results
`seqtools.spec.txt`	exact package files	the same environment again, on BinAC 2

Exercise: Your own environment

Goal: build an environment from a file and record what you got

1. Load Miniforge and add the `bioconda` channel
2. Write `~/seqtools.yml`: environment `seqtools` with `seqkit`
3. Create the environment from the file and activate it
4. Run `seqkit stats` on the example FASTQ files (they stay on PROJECT)
5. Save a snapshot with all versions to `~/seqtools.lock.yml`

Expected result: reads per file, and a snapshot listing `seqkit`

> **Help yourself:** the last slides, `conda env --help`

Bonus: which `seqkit` version did you get? Use `grep`

Quiz: What do you send?

A colleague asks for **exactly** the software versions behind the results in your paper.

1. Your hand-written `seqtools.yml`
2. A snapshot made with `conda env export` when you produced the results
3. A copy of the directory `~/.conda/envs/seqtools`
4. The list of `conda install` commands you remember typing

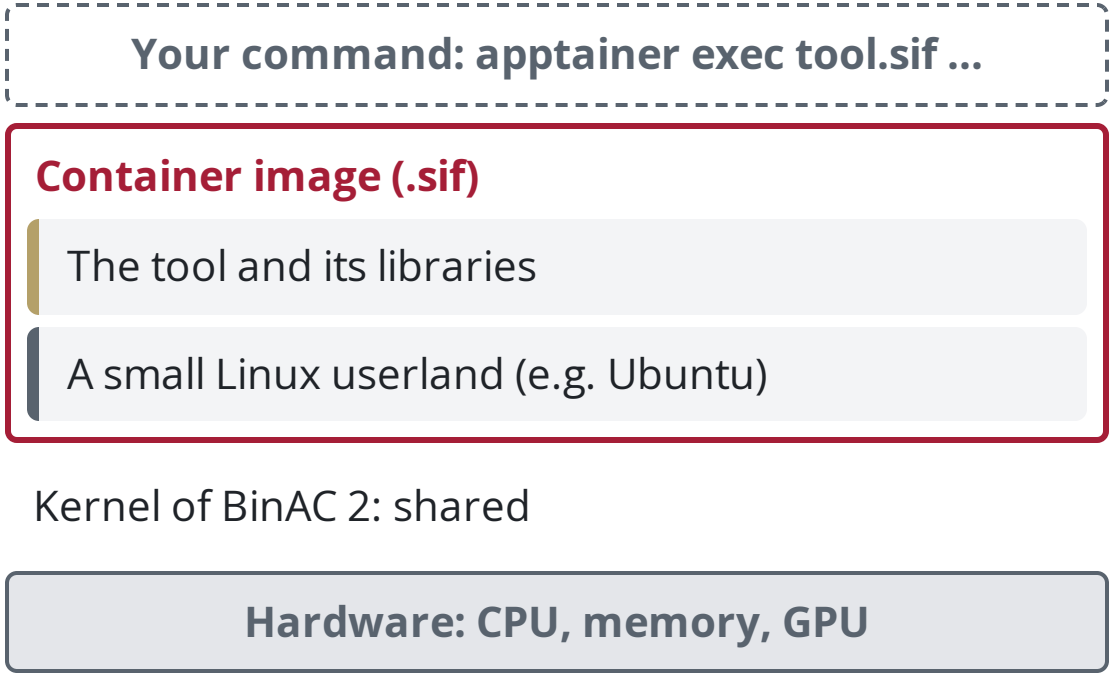
Clean up

```
conda env list                          # which environments exist?
du -sh ~/.conda/envs/*                  # how large is each?
conda env remove -n seqtools            # delete one environment
conda clean --all                       # empty the package cache
```

- Delete environments you no longer use: you can rebuild them from the file
- Delete with `conda env remove`, not `rm`: Conda keeps a list of your environments
- Broken environment? Do not repair it: delete it and create it again from the file

Containers: software in one file

- An **image** packs a whole software environment into **one file** (`.sif`)
- It runs on BinAC 2 like on any other Linux system: same versions, same results
- Many tools exist as images already: BioContainers, Docker Hub
- **Apptainer** runs them, without admin rights



Hello, container

```
cd "$(ws_find training)"
apptainer pull lolcow.sif docker://sylabsio/lolcow     # download and convert an image
apptainer run lolcow.sif                               # run its default command
apptainer exec lolcow.sif cat /etc/os-release          # run any command inside
apptainer shell lolcow.sif                             # look around inside, exit to leave
```

```
 ______________________________
< Sun Nov 15 14:02:11 CET 2026 >
 ------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

- `pull` once, then use the `.sif` file as often as you like
- Inside, you are still **you**: same user, your HOME and the current directory are there

Where images and the cache go

What	Where	Why
Images (`.sif`)	WORK or PROJECT, **not HOME**	large files, HOME is too small
Cache	`/pfs/10/project/apptainer_cache/$USER`	prepared for you, one per user

```
export APPTAINER_CACHEDIR=/pfs/10/project/apptainer_cache/$USER
du -sh "$APPTAINER_CACHEDIR"                    # how large is it?
apptainer cache clean                           # empty it
```

- `pull` downloads layers into the cache first, then builds the `.sif`
- Without `APPTAINER_CACHEDIR` the cache goes to `~/.apptainer`: HOME fills up

Containers in a job

```
#!/bin/bash
#SBATCH --partition=compute
#SBATCH --cpus-per-task=4
#SBATCH --time=01:00:00

img=/pfs/10/project/bw16f003/images/samtools_1.24.sif
apptainer exec "$img" samtools sort -@ "$SLURM_CPUS_PER_TASK" \
    -o sorted.bam input.bam
```

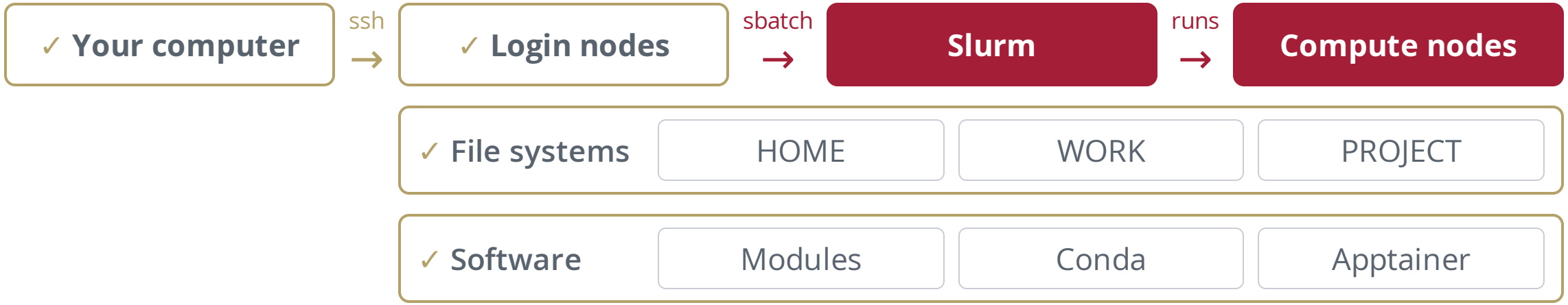
- Pull the image **before**, on the login node: the job only uses the file
- Your workspace and PROJECT are visible inside the container
- GPUs: `apptainer exec --nv` makes them visible inside
- Images from BioContainers: one image per tool and version, e.g. `quay.io/biocontainers/samtools`

Which source for my tool?

Step	Check	Then
1	`module avail NAME`	`module load`
2	`conda search NAME` (with `bioconda` set up)	a Conda environment
3	an image on BioContainers or Docker Hub	`apptainer exec`
4	nothing found	ask us: `hpcmaster@uni-tuebingen.de`

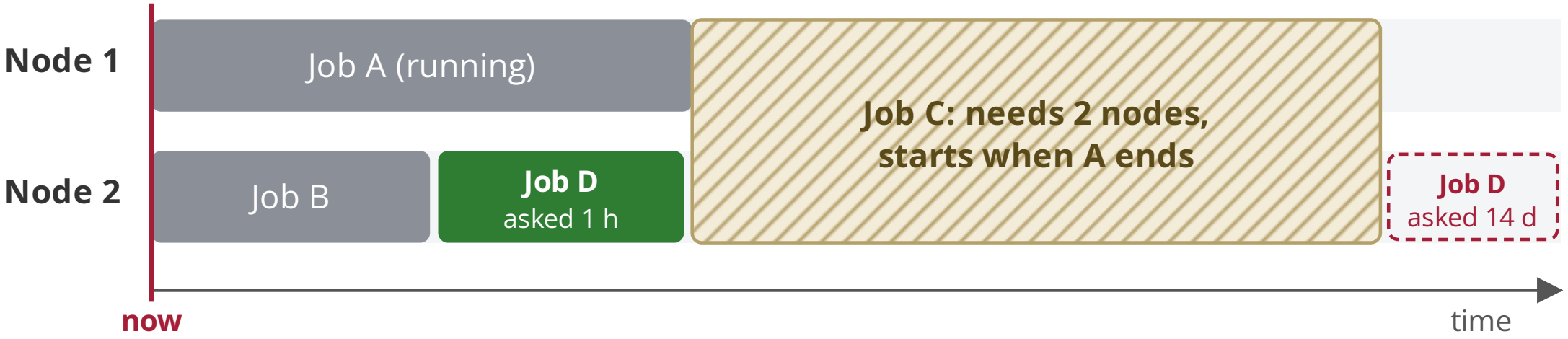
- Try in this order: the first hit is usually the least work
- Need a tool **many** users need? Ask for a module, even if Conda works

Batch jobs with Slurm



Why a batch system?

What if everybody just started their programs on the compute nodes?
Nodes overloaded, jobs slowing each other down, no fair share.



- You describe **what** you need, Slurm decides **when and where** it runs
- **Precise requests start sooner:** short jobs fill the gaps (backfilling)

Compute nodes

Type	Cores	Memory	GPUs	Local disk
Standard	64	512 GB		450 GB
High memory (SMP)	48 or 64	2 TB		14 TB
GPU A30	64	512 GB	2 × A30 (24 GB)	450 GB
GPU A100	64	512 GB	4 × A100 (80 GB)	14 TB
GPU H200	128	1.5 TB	4 × H200 (141 GB)	28 TB

Details: BinAC 2 hardware and architecture

- Hyperthreading: Slurm counts **2 CPUs per physical core**
- Jobs usually **share a node**, also its local disk (`$TMPDIR`)

Partitions

Partition	Nodes	Max. time	Notes
`compute`	CPU	14 days	default
`gpu`	A30, A100, H200	14 days	GPUs must be requested with `--gres`
`interactive`	4 CPU nodes	10 hours	max. 16 CPUs, 1 node
`long`	CPU, InfiniBand	30 days	max. 10 nodes

Details: BinAC 2 Slurm partitions

- Default if you request nothing: 1 CPU, 1 GB per CPU, **10 minutes**: always set time and memory

A job script

```
#!/bin/bash
#SBATCH --job-name=first_job          # name in the queue
#SBATCH --partition=compute           # where to run
#SBATCH --ntasks=1                    # one process
#SBATCH --cpus-per-task=2             # CPUs for this process
#SBATCH --mem=1G                      # memory for the whole job
#SBATCH --time=00:05:00               # max. run time, then the job is stopped
#SBATCH --output=%x-%j.out            # output file: name-jobid.out

echo "Job $SLURM_JOB_ID runs on:"
hostname
echo "Scratch directory: $TMPDIR"
sleep 120
```

- `#SBATCH` lines: requests to Slurm, **before** the first command
- The job starts in the directory where you ran `sbatch`

Submit, watch, cancel

```
sbatch first_job.sh              # submit, prints the job ID
squeue --me                      # your jobs
scontrol show job JOB_ID         # all details of one job
scancel JOB_ID                   # cancel
cat first_job-JOB_ID.out         # output, also while running
```

```
JOBID  PARTITION  NAME       USER      ST  TIME  NODES  NODELIST(REASON)
123456 compute    first_jo   tu_abc01  PD  0:00  1      (Priority)
123457 compute    first_jo   tu_abc01  R   0:42  1      node1-064
```

`ST` : `PD` pending (waiting), `R` running; `REASON` tells why it waits

Exercise: Your first job

Goal: write, submit, watch and cancel jobs

1. In your workspace, write the job script from the slide with `nano`
2. Submit it and watch it with `squeue --me`
3. Look at the output file while it runs
4. Submit it a second time and cancel this job

Expected result: one output file with node name and `$TMPDIR`

> **Help yourself:** `man sbatch`, `squeue --help`

Bonus: use `scontrol show job` to find out how much memory your job got

Quiz: Put the job script in order

```
A   samtools sort -@ 4 -o sorted.bam input.bam
B   #SBATCH --cpus-per-task=4
C   module load bio/samtools/1.24
D   #!/bin/bash
E   #SBATCH --time=01:00:00
```

Which order works?

1. D B E C A
2. B E D C A
3. D C B E A
4. D B E A C

Software in your job script

```
#!/bin/bash
#SBATCH --partition=compute
#SBATCH --time=00:30:00

module purge                            # clean start: nothing from your login shell
module load bio/samtools/1.24           # modules: full name with version
module load devel/miniforge
conda activate seqtools

samtools --version
seqkit version
```

- A job starts a **new shell**: load your software **in the script**, every time
- Not in `~/.bashrc`: the script alone shows what your job used
- `module purge` first: independent of what you loaded before `sbatch`

Interactive jobs

For testing and short interactive work on a compute node:

```
salloc --partition=interactive --cpus-per-task=4 --mem=8G --time=01:00:00
```

```
salloc: Granted job allocation 6705825
salloc: Nodes node1-001 are ready for job
[tu_abc01@node1-001 ~]$                    ← now on a compute node
```

- Same resource options as `sbatch`
- `exit` ends the job and frees the resources
- Max. 10 hours, 16 CPUs, one node

Resources: what to request

Option	Meaning
`--cpus-per-task=N`	CPUs for your program (threads)
`--mem=SIZE`	memory for the whole job, e.g. `8G`
`--time=HH:MM:SS`	maximum run time
`--gres=gpu:a30:1`	GPUs, in the `gpu` partition

- Slurm counts hyperthreads: **2 CPUs per physical core**
- One physical core per thread: for **n threads**, request `--cpus-per-task=2n`
- Tell your program how many threads to use: it does not know otherwise

```
samtools sort -@ "$SLURM_CPUS_PER_TASK" ...
```

- Too little memory or time: the job is killed. Too much: it waits longer

When a job fails

```
tail first_job-JOB_ID.out                                    # first look: the end of the output
sacct -j JOB_ID --format=JobID,State,Elapsed,MaxRSS,ExitCode
```

State	Meaning	Next time
TIMEOUT	the time limit was reached	more `--time`
OUT_OF_MEMORY	more memory used than `--mem`	more `--mem`, check with `seff`
FAILED	your program stopped with an error	read the output file
CANCELLED	stopped with `scancel`	

- Read the **end** of the output file: the error is usually in the last lines

Quiz: Why does my job wait?

```
$ squeue --me
JOBID  PARTITION  NAME     USER      ST  TIME  NODES  NODELIST(REASON)
123460 compute    big_job  tu_abc01  PD  0:00  1      (Resources)
```

What does `(Resources)` mean?

1. Your job script has an error
2. Not enough free resources right now: the job waits until they are free
3. You have to submit the job again with other resources
4. The partition is broken

GPU jobs

```
#!/bin/bash
#SBATCH --partition=gpu
#SBATCH --gres=gpu:a30:1                # type and number of GPUs
#SBATCH --cpus-per-task=8
#SBATCH --mem=32G
#SBATCH --time=02:00:00

nvidia-smi                              # shows the GPUs your job got
```

- GPU types: `a30`, `a100`, `h200`
- The `gpu` partition alone gives you **no GPU**: `--gres` is required
- Your job only sees the GPUs it requested
- Choose the smallest GPU that is enough: for your priority, an H200 counts like 40 CPUs, an A30 like 10

Many similar jobs: job arrays

```
#SBATCH --array=1-3                    # 3 tasks: 1, 2, 3
#SBATCH --array=1-100%10               # 100 tasks, max. 10 at once
```

- Each task runs the same script with a different `$SLURM_ARRAY_TASK_ID`
- Use it to pick the input:

  ```
  FILE="sample_${SLURM_ARRAY_TASK_ID}_R1.fastq.gz"
  ```

  `${...}` marks where the variable name ends
- Output per task: `#SBATCH --output=%x-%A_%a.out` (array ID, task ID)
- Better than 100 separate `sbatch` calls

Exercise: Count the reads, in parallel

Goal: process several samples with one array job

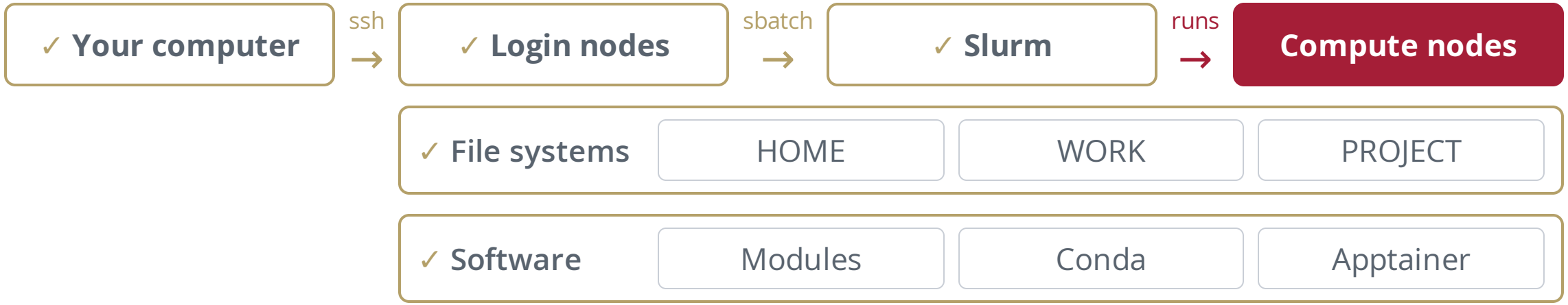
1. In your workspace, write `count.sh`: an array job with 3 tasks
2. Each task counts the lines of `sample_N_R1.fastq.gz` from the example data and writes the number to `results/count_N.txt`
3. Submit it and watch the 3 tasks in `squeue --me`

Expected result: `results/count_1.txt`, `count_2.txt`, `count_3.txt`

> **Help yourself:** the motivation slide from the start of the afternoon

Bonus: write the number of **reads** (lines / 4) instead, using `seqkit`

Efficient use of resources

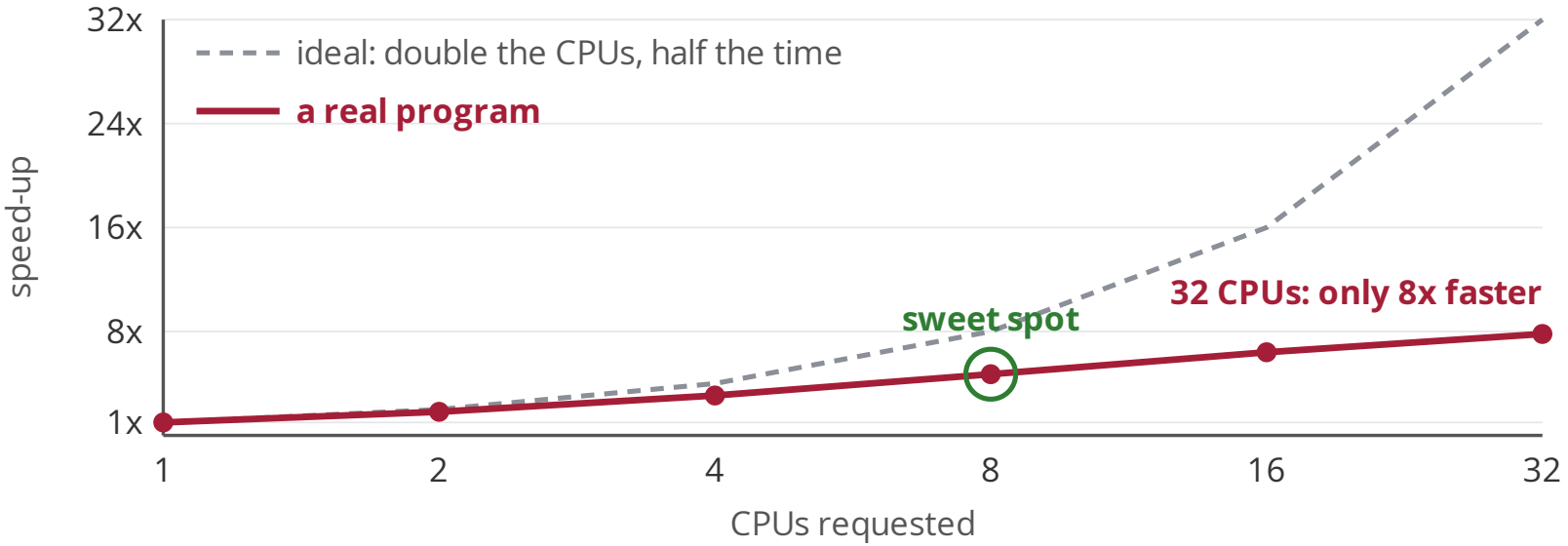


What you request is gone

```
$ seff 1093771
Nodes: 2
Cores per node: 128
CPU Utilized: 00:59:42
CPU Efficiency: 0.02% of 6914:08:00 core-walltime
Job Wall-clock time: 27:00:38
```

- 256 CPUs reserved for 27 hours, **one** CPU did the work
- Requested CPUs and memory are **blocked for everybody else**, used or not
- Your priority (fair share) is based on the CPUs and memory you **request**, not on what you use

More CPUs do not always help



- A program only uses several CPUs if it **can** and you **tell it to** (`-t` , `-@`)
- Find the **sweet spot**: test with 2, 4, 8, 16 CPUs on a small input
- Many independent inputs: a **job array** of small jobs beats one big job

Did my job use what it requested?

```
$ seff 1180637
State: COMPLETED (exit code 0)
Cores per node: 4
CPU Utilized: 00:02:11
CPU Efficiency: 37.64% of 00:05:48 core-walltime
Job Wall-clock time: 00:01:27
Memory Utilized: 9.00 MB
Memory Efficiency: 0.18% of 4.88 GB
```

- Low CPU efficiency: request fewer CPUs, or make your program use them
- Low memory efficiency: request less memory, your next job starts sooner
- Check every **new kind** of job once with `seff JOB_ID`, then adjust

Quiz: What would you change?

```
Cores per node: 16
CPU Efficiency: 6.10% of 16:00:00 core-walltime
Job Wall-clock time: 01:00:00
Memory Efficiency: 92.40% of 32.00 GB
```

1. Request more memory
2. Request fewer CPUs, or tell the program to use all 16
3. Request more time
4. Nothing, the job finished

Exercise: Check your jobs

Goal: compare requested and used resources

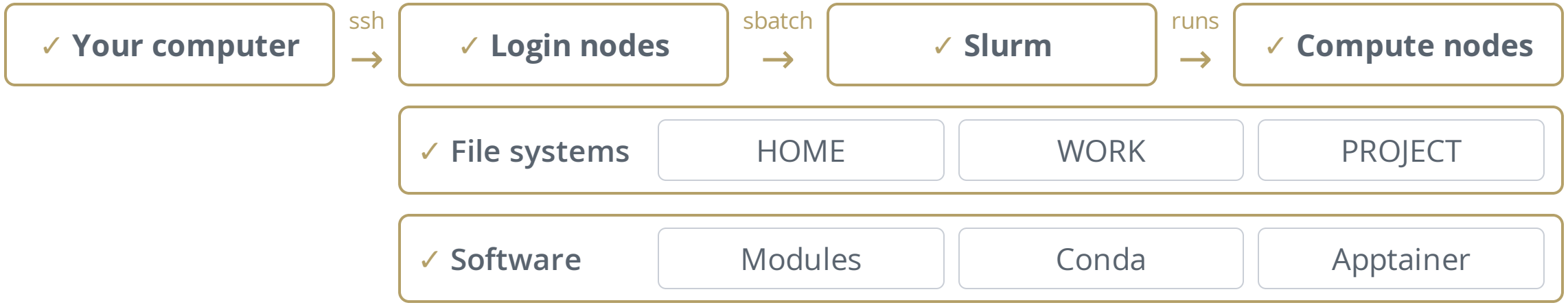
1. Run `seff` on your first job and on one task of your array job
2. Which values would you change for the next run?
3. List all your jobs of today with `sacct`

Expected result: CPU and memory efficiency for each job

> **Help yourself:** `sacct --help`, look for `--starttime`

Bonus: why is the CPU efficiency of `first_job.sh` so low?

Reproducibility



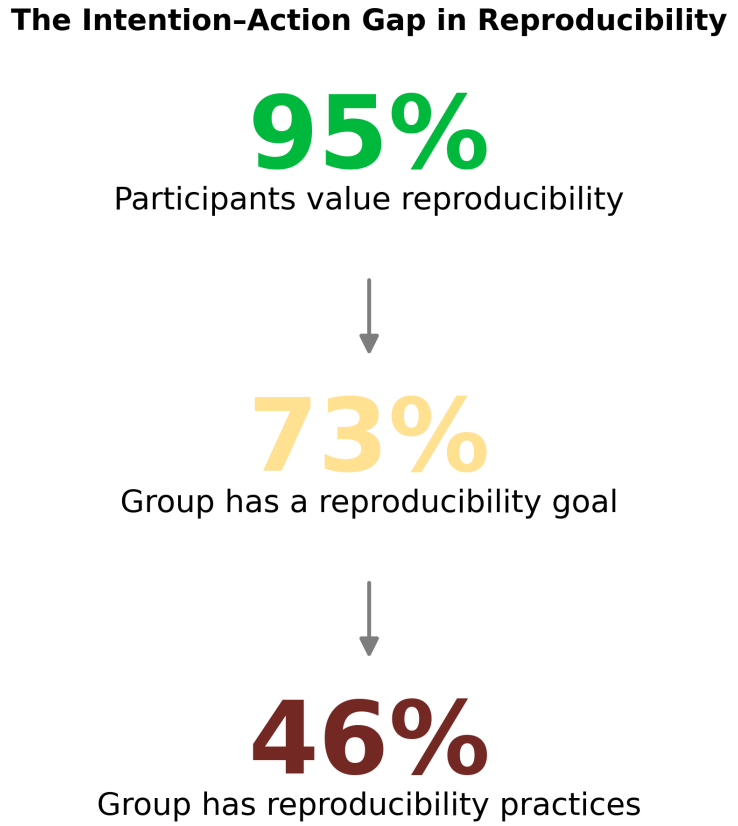
What is reproducibility?

	Same data	Different data
Same analysis	**Reproducible**	Replicable
Different analysis	Robust	Generalisable

The Turing Way: Guide for Reproducible Research, CC BY 4.0

- **Reproducible:** someone else, or you in a year, runs **your** code on **your** data and gets **your** results
- That is the minimum, and it is what this chapter is about: scripts, software versions, data

Everybody wants it, few do it



Survey of **422** users of HPC and cloud systems, 17 research fields:

- **95 %** say reproducibility matters
- only **46 %** say their group has concrete practices
- The gap is not motivation: it is **habits and tools**

Bartusch, Meesters, Sirvent (2026), survey poster, deRSE 2026, CC BY 4.0

Can you run it again in a year?

- **Scripts:** keep job scripts and configs in HOME (backup); learn version control (git) to keep their history, e.g. Software Carpentry: Version Control with Git
- **Software:** record versions
  - modules: write the full name, e.g. `bio/samtools/1.24`
  - Conda: a snapshot with `conda env export`, stored next to your results
  - containers: keep the image, note its version
- **Data:** never change raw data, keep it in PROJECT or an archive
- **Workflow:** many steps? Consider a workflow management system
- Not all or nothing: every step helps

Many steps? Workflow managers

Snakemake and **Nextflow** describe your whole analysis as code:

- **Versioned:** pipeline code and configs live in git, so every result belongs to a known version
- **Orchestrated:** the tool knows which step needs which input, submits the steps as Slurm jobs,
  runs them in parallel and reruns only what changed
- Software per step from Conda or Apptainer, ready-made pipelines: nf-core, Snakemake Workflow Catalog

Record what a run did, as an RO-Crate: Snakemake report plugin (with contributions from us), nf-prov for Nextflow

Now you can read this

```
#!/bin/bash
#SBATCH --partition=compute               # Slurm: requests, before the first command
#SBATCH --cpus-per-task=8                 # resources: CPUs, memory, time
#SBATCH --mem=16G
#SBATCH --time=02:00:00
#SBATCH --array=1-3                       # job array: 3 tasks
#SBATCH --output=logs/map_reads_%A_%a.out   # one log per task

module load bio/bowtie2/2.5.4             # software from modules
module load bio/samtools/1.24
ws=$(ws_find training)                    # data in a workspace
sample="sample_${SLURM_ARRAY_TASK_ID}"    # task ID picks the input

cd "$TMPDIR"                              # heavy I/O on the local disk
bowtie2 -p "$SLURM_CPUS_PER_TASK" -x "$ws/ref/genome" \
    -1 "$ws/fastq/${sample}_R1.fastq.gz" \
    -2 "$ws/fastq/${sample}_R2.fastq.gz" \
  | samtools sort -@ 4 -o "${sample}.bam" -

samtools index "${sample}.bam"
cp "${sample}.bam" "${sample}.bam.bai" "$ws/results/"   # results back
```

Summary

- Login nodes for preparation, compute nodes through Slurm
- HOME for scripts, workspaces for active data, PROJECT for shared data, `$TMPDIR` inside jobs
- Software from modules, Conda, or containers
- Job script: `#SBATCH` requests, then commands; `sbatch`, `squeue --me`, `scancel`
- Request what you need, check with `seff`
- Write down how you got your results

Getting help

- Documentation: wiki.bwhpc.de/e/BinAC2
- Support: `hpcmaster@uni-tuebingen.de` or the ticket system bw-support.scc.kit.edu
- Announcements: mailing list `binac_announce` (you are subscribed on registration)
- More: Tiger Teams for intensive support, see the introduction