Making Data Fun Again

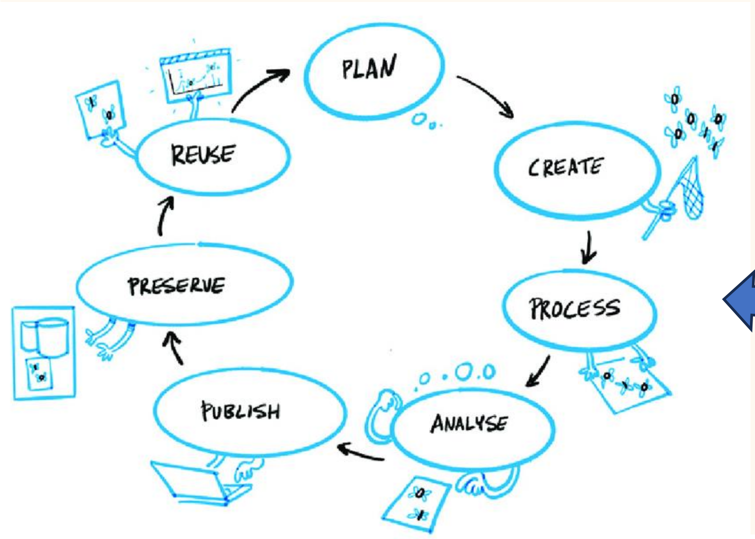
Extending EESSI to improve Research Data Management

Thomas Röblitz, University of Bergen

FOSDEM'25, HPC devroom, Feb 2nd

What is Research Data (Management)?

Research data is any data (consumed or produced) in research.

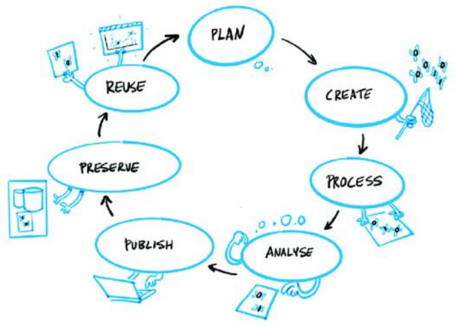


By Patrick Hochstenbach, Ghent University

Research Data Management comprises all kinds of activities to “organise” research data with the goal to enable its reuse.

Dilemma

Structure



workflows

DMPs

?

Creative Chaos (the fun part)

The actual problem with Research Data Management

- not acknowledged in hiring processes and proposal evaluations
- underfunded (varying support from organizations)
- costs time

➔ researchers do the minimum to satisfy (funders‘) requirements

- Plus, RDM is just one issue for researchers using IT…

What in IT do researchers struggle with?

- many IT systems: laptop, cloud, HPC
- use many software packages
- develop code for increasingly heterogeneous architectures

- manage their own *virtual data infrastructure*
  - low-level (data) management operations to move data between systems
  - manage storage spaces with different performance and quota

How EESSI helps already and what could be next?

many IT systems: laptop, cloud, HPC

easy access to many software packages

develop code for increasingly heterogeneous architectures

?

- manage their own *virtual data infrastructure*
  - low-level (data) management operations to move data between systems
  - manage storage spaces with different performance and quota

“Extending” EESSI to make data handling easier

Don’t limit creativity

Avoid (low-level) data management operations

"extend" EESSI

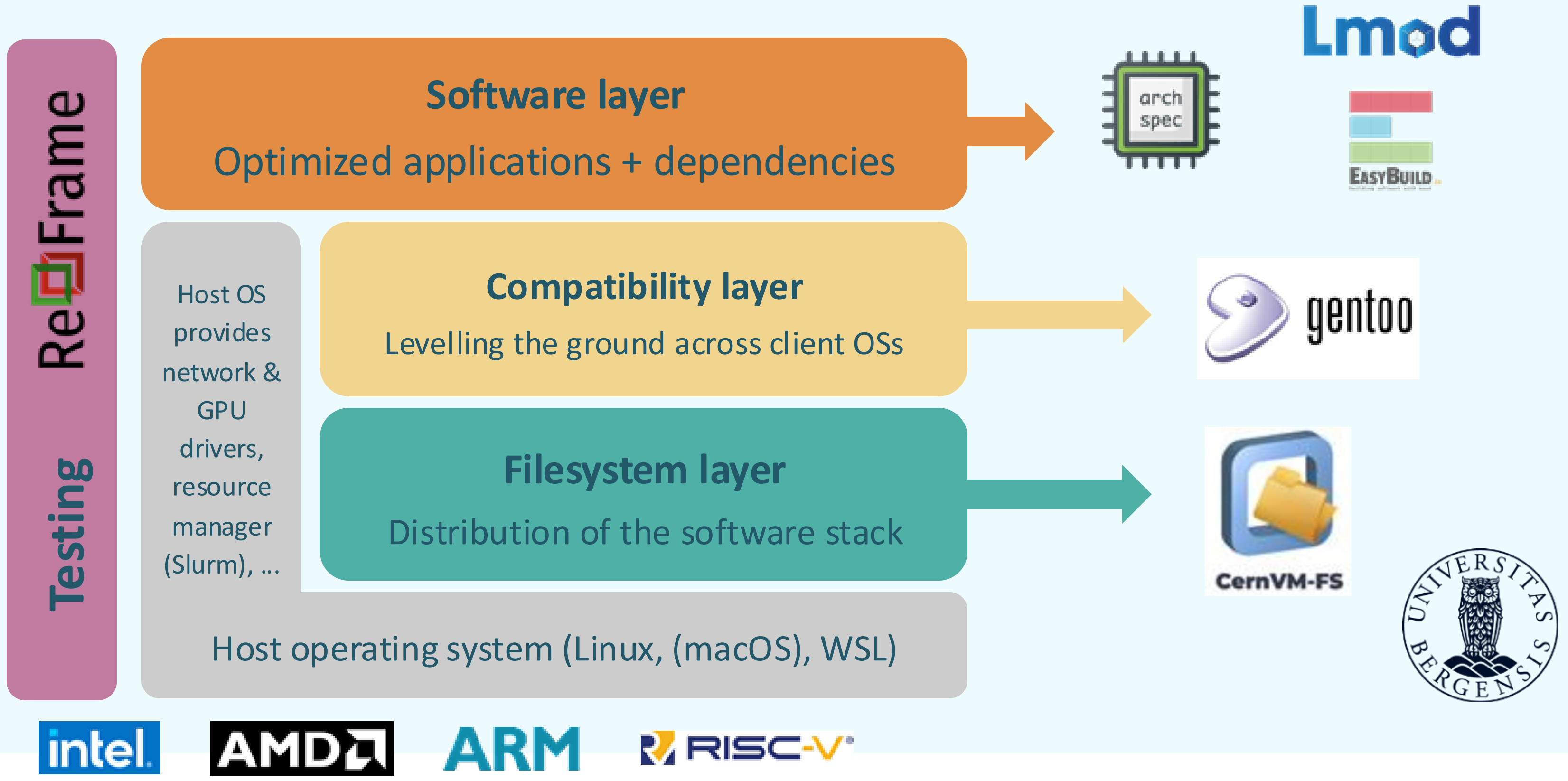
everyone should be already using EESSI

EESSI provides a good basis

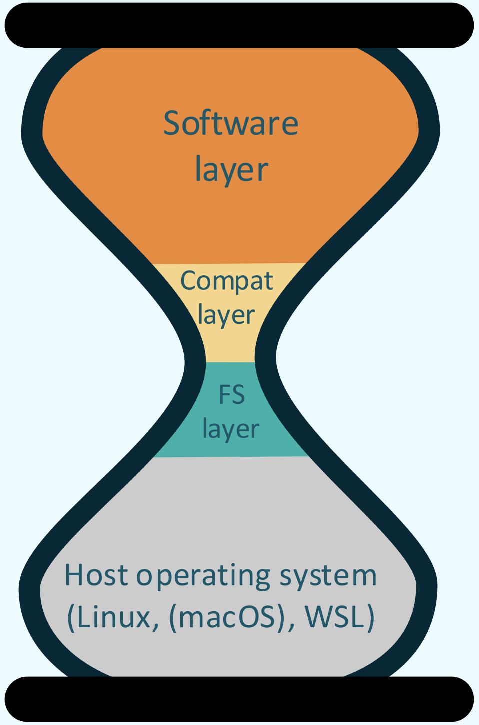
EESSI in a nutshell

- **On-demand streaming** of **optimized** scientific software installations
- **Works on any Linux distribution** thanks to EESSI compat layer
- **Uniform software stack** across various systems: laptop, HPC, cloud, ...
- Community-oriented: **let's tackle the challenges we see together!**

EESSI's layered architecture


ReFrame
Testing
Software layer
Optimized applications + dependencies
arch spec
Lmod
EasyBuild
Host OS provides network & GPU drivers, resource manager (Slurm), ...
Compatibility layer
Levelling the ground across client OSs
gentoo
Filesystem layer
Distribution of the software stack
CernVM-FS
Host operating system (Linux, (macOS), WSL)
intel
AMD
ARM
RISC-V
UNIVERSITAS BERGENSIS

How can we "extend" EESSI?



- don't want to rebuild every software installation
- *better: modify or augment a core component*
- assumption: most data accesses pass through compat layer
- key data access functions:
    - `*open*`: `open`, `fopen`, `openat`, `fopenat`, `open64`, `fopen64`, `freopen`
    - `read`, `write`
    - `close`, `fclose`
- all part of GLIBC: changes there apply to all/most software

How can we change GLIBC functions?

- Directly change GLIBC
  - pros: transparent for users, no change needed
  - cons: is always on whether needed/wanted or not; may have undesired consequences; need to change a shipped installation
- Wrap functions and use `$LD_PRELOAD`
  - pros: keep default GLIBC; easily switched on/off by users
  - cons: need to adjust "every" run for full coverage; may create conflicts if set globally
- For prototype: wrap functions and use `$LD_PRELOAD`

What could we do with wrapping GLIBC functions?

log information about certain calls (open, exec, ...)

post-process logs to create data flow graphs

enable using remote data "directly"

no manual download of data before processing it

define a virtual data infrastructure

declare which data is needed

runtime ensures that data is available

Idea 1: log information about certain GLIBC calls

- Example: `open()`

```
int open(const char* pathname, int flags, mode_t mode) {
    printf("vdi_log: call %s with '%s', %d, %d\n",
           __func__, pathname, flags, mode);
    int (*actual_open)() = dlsym(RTDL_NEXT, __func__);
    return actual_open(pathname, flags, mode);
}
```

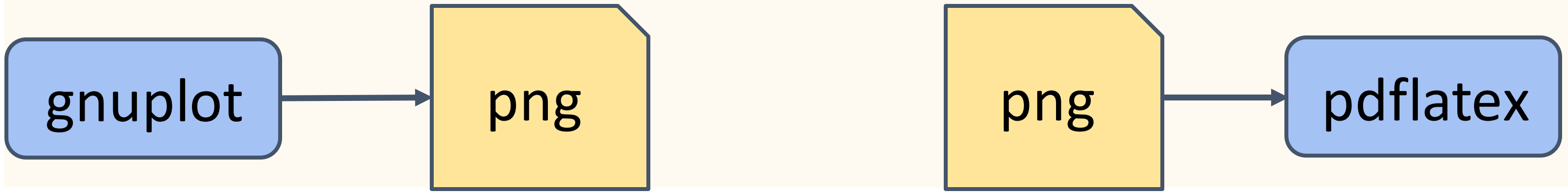
- `gcc -fPIC -shared -o libvdi.so vdi.c -ldl`
- `LD_PRELOAD=libvdi.so cat /etc/os-release`

Idea 1: log information about certain GLIBC calls

- Log lines

```
timestamp hostname user $HOME pid ppid pgid
$PWD program argv starttime elapsed call
call_args
```

- analyse logs to construct data flow graph
- each line produces

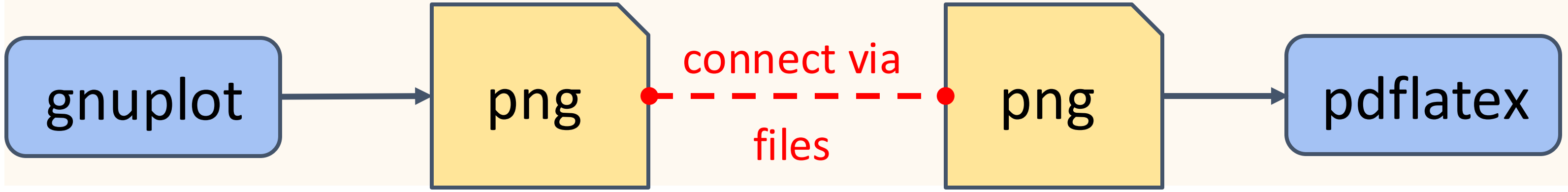


Idea 1: log information about certain GLIBC calls

- Log lines

```
timestamp hostname user $HOME pid ppid pgid
$PWD program argv starttime elapsed call
call_args
```

- analyse logs to construct data flow graph
- each line produces



Idea 1: log information about certain GLIBC calls

- Use cases
  - automatically describe how results (data/figs) were produced
  - automatically generate "workflow" descriptions
  - detect if some (input) files were not used
  - create a timeline of the work (travel back finally possible)
  - improve program start-up times (e.g., Spindle)
  - EESSI: ensure that build process used the correct data/libs
  - EESSI: determine files to pre-load caches

Idea 2: enable access to remote data "directly"

Pattern: a file/dataset is downloaded, then it is used

Goal: avoid the *manual* download step

Benefits:

makes explicit which data source has been used

if data has a PID -> can get metadata for dataset

Idea 2: enable using remote data “directly”

- Example: `open()`

```
int open(const char *pathname, int flags, ...) {
  if (is_url(pathname)) {
    if (download(pathname, &local_path) == 0) {
      return actual_open(local_path, flags);
    }
  }
}
```

Idea 2: enable using remote data "directly"

- **Example:** `open()`
- `gcc -fPIC -shared -o libvdi.so vdi.c -ldl -lcurl`
- **(optional)** `export VDI_DOWNLOAD_BASE=/project/vdi_example`
    - **default:** `/tmp/$USER/vdi/downloads`
- `vdi run wc -l https://zenodo.org/records/13830932/files/ruthalicia_longipes_spades_01.fasta.3.fasta?download=1`

Idea 2: enable using remote data “directly”

"bit" long to write - could support alternatives:

`doi://10.5281/zenodo.13830932` *or*

`zenodo://13830932` *or*

`zenodo://13830932?ruthalicia_longipes_spades_01.fasta.3.fasta`

-
  ```
  vdi run wc -l
  https://zenodo.org/records/13830932/files/ruthalicia_longipes_spades_01.fasta.3.fasta?download=1
  ```

Idea 2: enable using remote data “directly”

- works surprisingly well - though not with all commands such as `tar`
- only for *read* access
- *write* could work too:
  - requires some kind of API at receiver (zenodo provides an API)
  - upload when file closed
- should be possible to optimize
  - only download once *or* remove download when file closed
  - download a dataset early to prefetch it
- using PIDs (DOI, etc) allows to obtain metadata

Idea 3: define a virtual data infrastructure (vdi)

- What if all data is accessed via some prefix/namespace?

```
vdi://namespace/path_or_label/file[?params]
```

- A researcher would have to...
  - give the virtual data infrastructure a name (namespace)
  - register data/files/URLs with it

Idea 3: define a virtual data infrastructure (vdi)

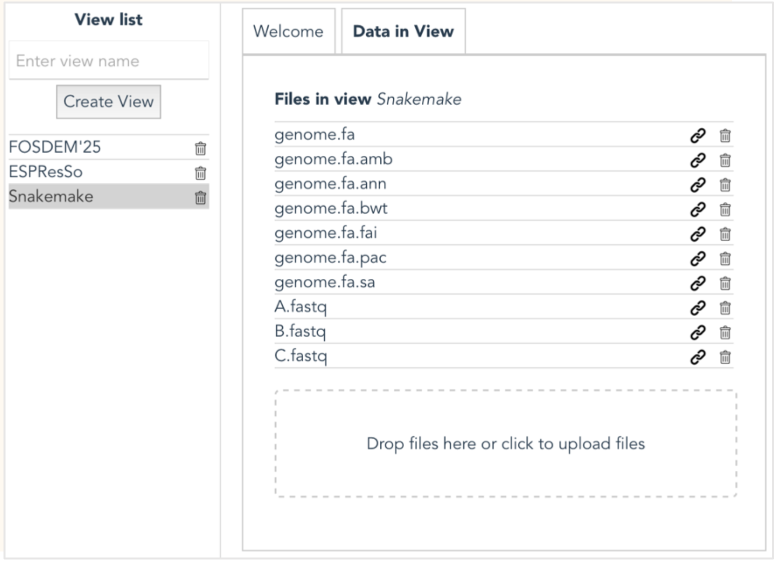
- Runtime could then
  - figure out where the file is stored (locally / remotely),
  - redirect or load the file,
  - upload files to a sync server or service,
  - create copies,
  - give access to others,
  - ...

Idea 3: define a virtual data infrastructure (vdi)

- First step: create namespace, add files, obtain URL, ...
- Commands added

```
vdi view create <name>
vdi view list
vdi view upload <name> <file>
vdi view files <name>
vdi view geturl <name> <file>
```

Manage namespaces via GUI



Files in view *Snakemake*

genome.fa
genome.fa.amb
genome.fa.ann
genome.fa.bwt
genome.fa.fai
genome.fa.pac
genome.fa.sa
A.fastq
B.fastq
C.fastq

Idea 3: define a virtual data infrastructure (vdi)

- Example remote data access

```
vdi run head https://vdi.nessi.no/download/Snakemake/genome.fa

>I dna_rm:chromosome chromosome:R64-1-1:I:1:230218:1 REF
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNTCCTAACACTACCCTAACACAGCCCTAATCTAACCCTGGCCAACCTGTCTCTCAACTT
ACCCTCCATTACCCTGCCTCCACTCGTTACCCTGTCCCATTCAACCATACCACTCCGAAC
CACCATCCATCCCTCTACTTACTACCACTCACCCACCGTTACCCTCCAATTACCCATATC
CAACCCACTGCCACTTACCCTACCATTACCCTACCATCCACCATGACCTACTCACCATAC
TGTTCTTCTACCCACCATATTGAAACGCTAACAAATGATCGTAAATAACACACACGTGCT
TACCCTACCACTTTATACCACCACCACATGCCATACTCACCCTCACTTGTATACTGATTT
TACGTACGCACACGGATGCTACAGTATATACCATCTCAAACTTACCCTACTCTCAGATTC
CACTTCACTCCATGGCCCATCTCTCACTGAATCAGTACCAAATGCACTCACATCATTATG
```

- Basic proof of concept

Status

- prototype implementation
- `vdi` cli command and wrapper library
  - open source, builds and installs in less than 1 min
  - could be shipped with EESSI
- vdi "server" (frontend, API, backend/storage)
  - supports easy creation of data flow graphs
  - supports basic virtual data infrastructure

Outlook

- data flow graph
    - exports: Nextflow, snakemake, LaTeX, ...
    - intercept more functions: *exec*
    - improve performance
- virtual data infrastructure
    - enable use of backends: iRODs, *Cloud, ...
    - *runtime that actively manages data in the background*

Summary

- Working with research data is not so easy
- Two tools
  - logging data accesses and create data flow graphs
  - effortlessly accessing remote data
- Early prototype: `vdi` client, wrapper library, frontend & backend

➔ Researchers can focus on working with the data … tools take care of the data management

Resources

- Original report describing the ideas: https://doi.org/10.5281/zenodo.14788711
- Follow-up case study with iRODS: https://doi.org/10.7494/csci.2012.13.4.21
- EESSI – European Environment for Scientific Software Installations
    - docs - Join EESSI Slack - Paper (open access) - github
- MultiXscale – EuroHPC Centre of Excellence (main development of EESSI)
- Code developed for this talk: https://github.com/virtual-data-infrastructure